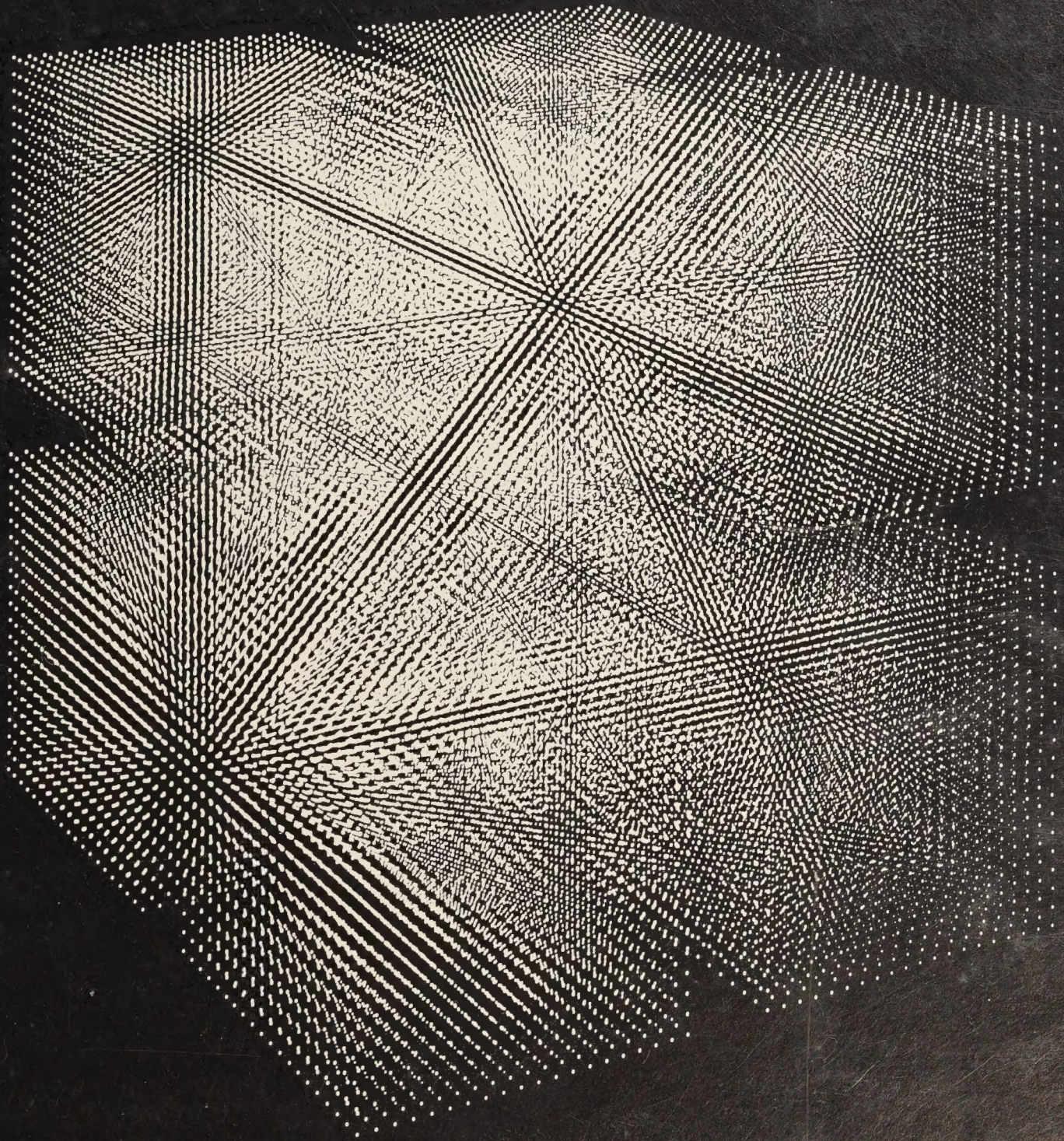
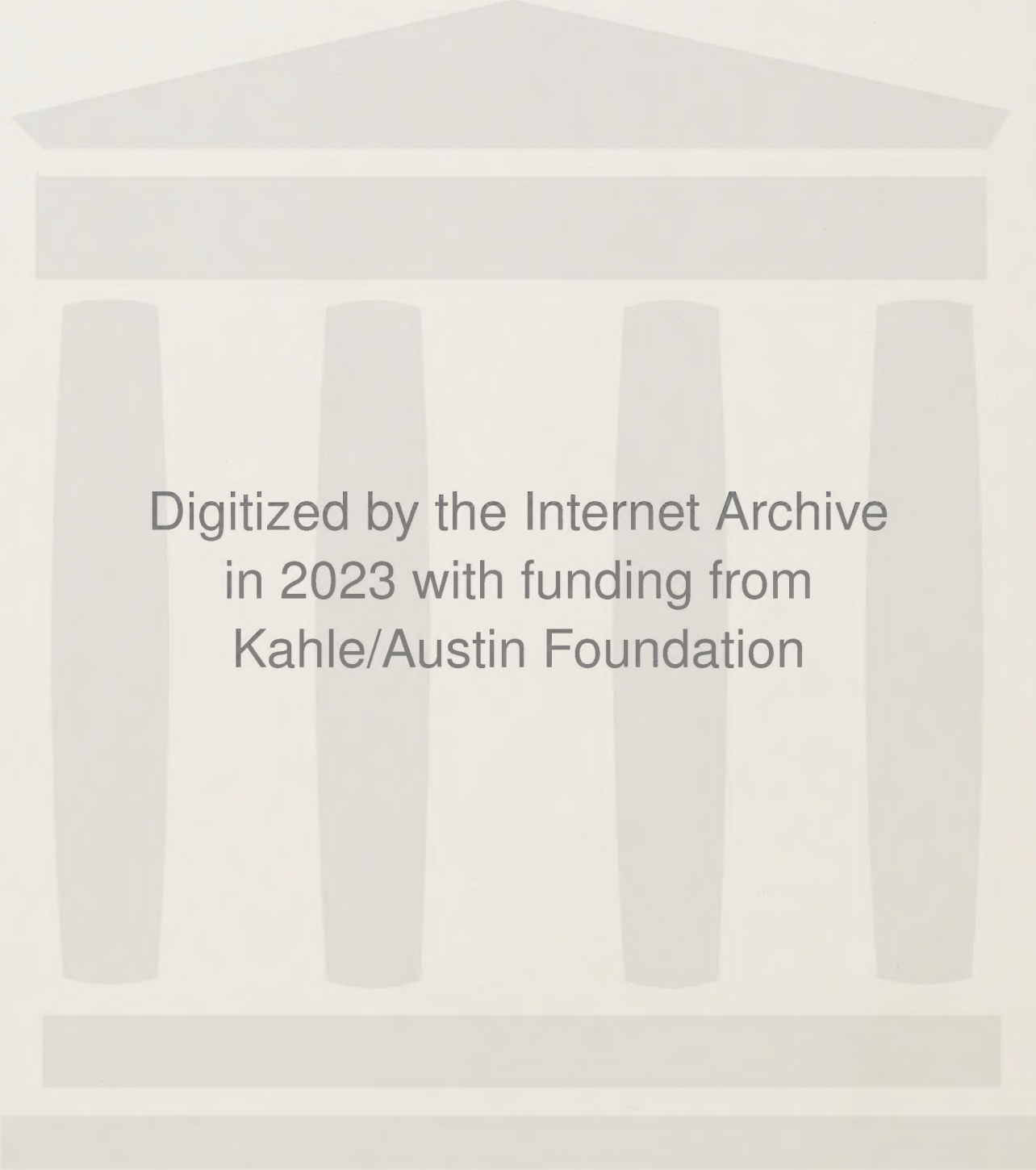


Thinking Machines Corporation

**The
Connection Machine[®]
System**





Digitized by the Internet Archive
in 2023 with funding from
Kahle/Austin Foundation

<https://archive.org/details/thnkingmachinesc01unse>

Guide to Documentation

Documentation Set Contents

Volume I *Guide to Documentation*

Connection Machine User's Guide: Using the Symbolics 3600 as a Front End

*The Essential *LISP Manual*

Connection Machine Parallel Instruction Set (PARIS): The LISP Implementation

User Contributed Software

Release Notes

Volume II *Guide to Documentation*

Connection Machine User's Guide: Using a UNIX System Front End

Connection Machine Parallel Instruction Set (PARIS): The C Implementation

Introduction to Data Level Parallelism

Overview

The languages used to program the Connection Machine are *LISP, the LISP implementation of PARIS, and the C implementation of PARIS. The table below shows the programming environments and front ends to the Connection Machine from which one programs. The diagram on the facing page and "What To Read" below show the manuals relating to the particular languages and environments.

Language	Description, Programming Environment
*LISP	Based on the LISP language Used within a LISP environment
PARIS	Parallel Instruction Set The Connection Machine's assembly language
PARIS (LISP): PARIS (C):	Used within a LISP environment Used on a UNIX system

Programming Environment	Front End on Which Environment Is Currently Available
LISP	Symbolics LISP Machine Lucid LISP on a UNIX operating system
UNIX	Digital Equipment Corporation VAX front end, running the ULTRIX operating system (ULTRIX is Digital's implementation of UNIX)

What To Read

Of Interest to All Connection Machine Users

Begin with the User's Guide appropriate for you, and continue with:
Release Notes (Volume I)
Introduction to Data Level Parallelism (Volume II)

Applies to Either LISP Environment

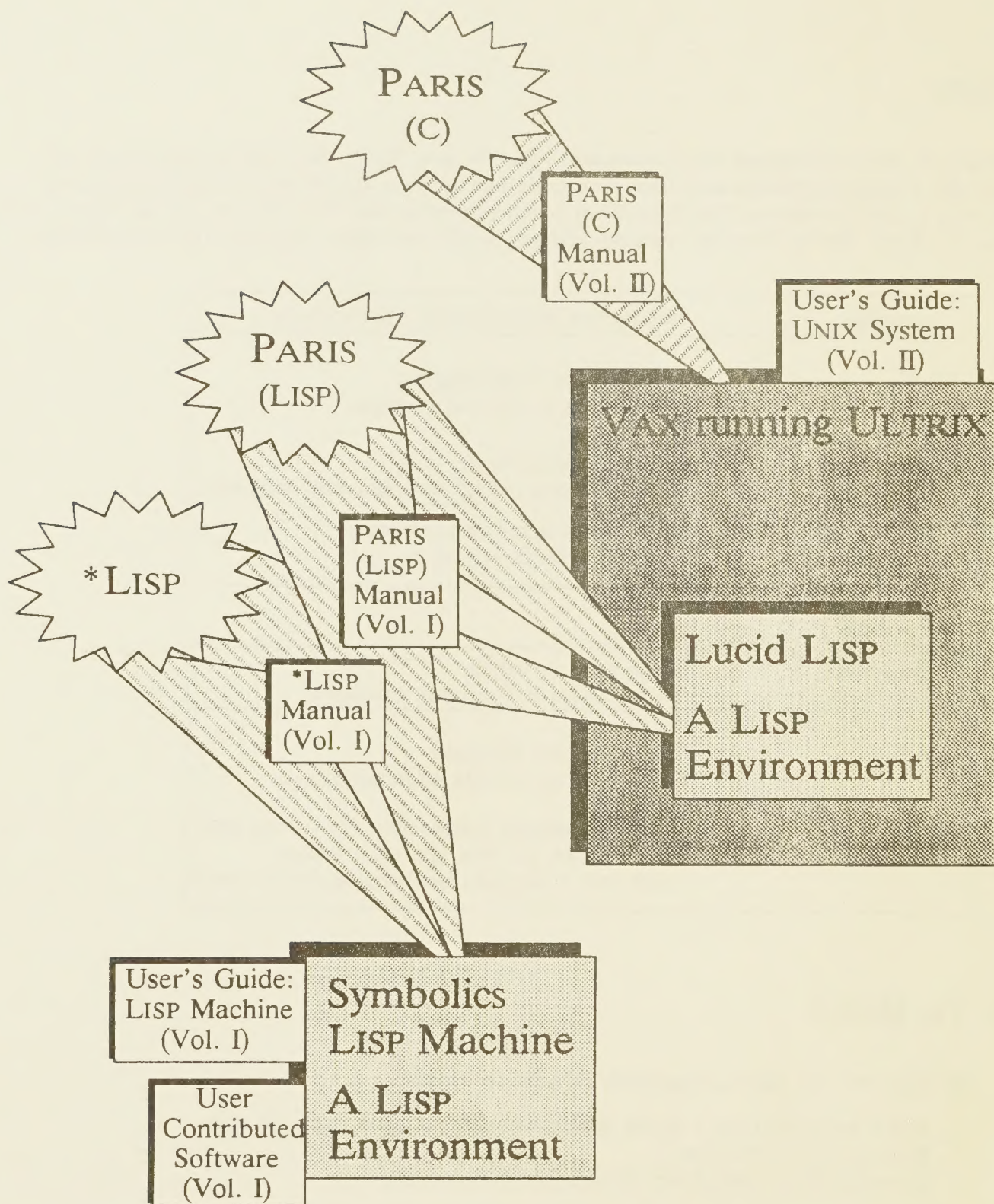
*The Essential *LISP Manual* (Volume I)
Parallel Instruction Set (PARIS): The LISP Implementation (Volume I)

Specific to a LISP Machine Front End

User's Guide: Using the Symbolics 3600 As a Front End (Volume I)
User Contributed Software (Volume I)

Specific to a UNIX System Front End

User's Guide: Using a UNIX System Front End (Volume II)
Parallel Instruction Set (PARIS): The C Implementation (Volume II)



Languages, programming environments, front ends, and manuals for Connection Machine programming, as described on facing page.

Thinking Machines Corporation

Connection Machine[®] User's Guide
Using the Symbolics 3600 as a Front End

March 1987

© 1986 Thinking Machines Corporation
All Rights Reserved

This notice is intended as a precaution against inadvertent publication and does not constitute an admission or acknowledgement that publication has occurred or constitute a waiver of confidentiality. The information and concepts described in this document are the proprietary and confidential property of Thinking Machines Corporation.

Document number 1-0001-1-6

© 1987 Thinking Machines Corporation.

"Connection Machine" is a registered trademark of Thinking Machines Corporation.

"PARIS" and "*Lisp" are trademarks of Thinking Machines Corporation.

"Symbolics 3600" is a trademark of Symbolics, Inc.

The information in this document is subject to change without notice and should not be construed as a commitment by Thinking Machines Corporation. Thinking Machines Corporation reserves the right to make changes to any products described herein to improve functioning or design. Although the information in this document has been reviewed and is believed to be reliable, Thinking Machines Corporation does not assume responsibility or liability for any errors that may appear in this document. Thinking Machines Corporation does not assume any liability arising from the application or use of any information or product described herein.

CONTENTS

i

Contents

1 Purpose Of This Document	1
2 Loading Connection Machine Software	1
3 Using The Connection Machine System From PARIS	2
4 Using The Connection Machine System From *Lisp	5
5 Basic Connection Machine Operation	6
5.1 Using The Control Panel On A 16K Machine	6
5.2 Using The Control Panel On A 64K Machine	7
6 Connection Machine Programming	9
7 Diagnostics	9
8 User Support	10

List of Figures

1	Menu For Loading Connection Machine Software	1
2	Front Panel for 16,384 Processor Machine	7
3	Front Panel for 65,536 Processor Machine	8

1 Purpose Of This Document

This guide is written for people who are going to program the Connection Machine system, using the Symbolics 3600 as a front end. It briefly outlines what is necessary to load the Connection Machine software onto the 3600, and to attach and use Connection Machine hardware from the 3600. It assumes that the reader is an experienced 3600 user, thoroughly familiar with all basic operations procedures.

2 Loading Connection Machine Software

The Connection Machine software is loaded into a standard Symbolics release world. Typically during software installation, bands containing the various software combinations will be made, in which case the method described below will not be necessary. For example, ***Lisp on PARIS hardware** might be loaded by booting with a **cm.boot** file. Similarly, ***Lisp on the PARIS simulator** might be booted with **starsim-on-PARIS.boot**, and the ***Lisp simulator** with **starsim.boot**. Check with the person responsible for your site administration. If the bands have not been made, the software is loaded with the following two functions:

```
(make-system 'cm-software-version-f5 :noconfirm)
(setup-cm-environment)
```

The function **setup-cm-environment** displays a menu of possible software to load (Figure 1). Selecting the menu item that corresponds to the language and the execution environment loads in all of the necessary software. Loading software is a one-step process.

Select the combination of software and hardware you want
*LISP on the PARIS SIMULATOR
Just the PARIS SIMULATOR
The *LISP SIMULATOR
PARIS on hardware
*Lisp on hardware

Figure 1: Menu For Loading Connection Machine Software

The two user facilities for programming the Connection Machine: ***Lisp**, a high level language, and **PARIS**, an assembly language for the Connection Machine system. Both ***Lisp** and **PARIS** can execute either on physical Connection Machine hardware, or on a

simulator that runs on the 3600. The simulators are much slower than actual hardware, and are primarily useful for testing Connection Machine programs on a small amount of data when there is no Connection Machine hardware available.

There are two ways to load a *Lisp simulator. The faster simulator is loaded by selecting the menu item **The *LISP SIMULATOR**. The other menu entry, ***LISP on the PARIS SIMULATOR**, loads a much slower simulator that is only used to simulate programs that mix *Lisp and PARIS code.

After loading the Connection Machine software, the next step is to connect the front end system to Connection Machine processors, and initialize them for use.

3 Using The Connection Machine System From PARIS

Before PARIS operations can be used to control a Connection Machine system, two things must be done:

- First, Connection Machine processors must be allocated for use.

Connection Machine processors are allocated for use in PARIS with the `cm:attach` operation. If actual Connection Machine hardware is to be used, `cm:attach` allocates physical processors and appropriately configures them as virtual processors. If the PARIS simulator is to be used, `cm:attach` configures the simulator to simulate the specified number of virtual processors.

- Second, the Connection Machine processors and memory must be initialized for use. The `cm:cold-boot` operation does this.

Typically the PARIS software need be loaded into a Lisp environment only once. It is possible to attach processors for use, release them (using the `cm:detach` operation), and then later attach processors again (possibly the same processors or different ones). Detaching processors makes them available for use by other front-end computers, but of course their state and memory contents are then lost.

While processors remain attached it may be desirable to reinitialize the state of the processors. One may use `cm:cold-boot` at any time to reset the processors to a standard initial state. It is a good idea for an application to call `cm:cold-boot` before doing anything else.

There is also an operation `cm:warm-boot` to reset the processors while preserving the contents memory. If a computation using the Connection Machine system should be interrupted at some point and not continued, it is usually best to call either `cm:cold-boot` or `cm:warm-boot` before invoking any other PARIS operation.

Here is a transcript of a short session on a Symbolics 3600 illustrating the use of the Connection Machine system. In practice, of course, one does not always type in PARIS

instructions one at a time from the keyboard, but writes large programs that call PARIS operations. This sample session does demonstrate interactive debugging in a Lisp-based system, as well as illustrate the use of `cm:attach`, `cm:cold-boot`, and `cm:detach`.

```
(make-system 'cm-software-version-f5 :noconfirm) (setup-cm-environment)
```

This causes a menu to pop up; using the mouse, the user (call him George) chooses to use PARIS with hardware. (Alternatively, the user could have booted the front-end with a band containing that software selection.) Now hardware must be allocated.

```
(cm:attach :16k)
NIL
```

Now George is attached to 16,384 physical processors.

```
(cm:finger)
```

Connection Machine System Rainbow			Physical size: 64K processors with 4K RAM
Jane	Red	Microcontroller Port (3)	<-- 16384 physical processors
Judy	Blue	Not Attached to a Port	
George	Yellow	Microcontroller Port (1)	<-- 16384 physical processors
Elroy	Green	Microcontroller Port (0)	<-- 16384 physical processors

From what `cm:finger` prints we can see that George is using the front-end computer named Yellow, that the Connection Machine system is named Rainbow and has a total of 65,536 physical processors, and that two other users, Jane and Elroy, are also attached to portions of the Connection Machine system. User Judy is logged in to the front-end computer named Blue, but is not attached to any Connection Machine hardware.

```
(cm:cold-boot)
16384
16384
3579
```

George cold-boots his portion of the Connection Machine system. He allows the number of virtual processors to default to the number of physical processors. There are 3579 bits of memory per virtual processor available to the user.

```
(cm:my-cube-address 0)
NIL
```

```
(cm:set-context)
```

NIL

```
(cm:global-add 0 cm:*cube-address-length*)
<error>
```

George causes every processor to put its own cube address into memory starting at location 0. He then tries to calculate the sum of these addresses, thereby summing the integers from 0 to 16383, but an error occurs. (The details of the error message are system-dependent and are not shown here.) The problem is that the `cm:global-add` operation requires the use of temporary scratch space in the gap, and no gap space has been allocated since the `cm:cold-boot` operation, which resets the stack, gap, and upper data areas to be empty.

```
(cm:set-stack-limit (- (cm:get-stack-limit) 50))
3528
```

```
(cm:global-add 0 cm:*cube-address-length*)
-8192
```

George allocates 50 bits of gap area and tries again to take the sum. The result is surprising: a negative number. Aha! Cube addresses should be treated as unsigned integers; treating them as signed causes half of them to be considered negative.

```
(cm:global-unsigned-add 0 cm:*cube-address-length*)
134209536
```

Using an unsigned summation produces a more reasonable result. But is it correct?

```
(/ (* 16383 16384) 2)
134209536
```

Yes, that is the right answer.

```
(cm:cold-boot)
16384
16384
3579
```

```
(cm:global-unsigned-add 0 cm:*cube-address-length*)
136
```

Calling `cm:cold-boot` can cause the contents of memory to be lost.


```
(cm:my-cube-address 0)
NIL
```

```
(cm:global-add 0 cm:*cube-address-length*)
134209536
```

After reconstructing the cube address values, George then gets the correct answer again for the sum.

At this point Jane politely requests that George and Elroy detach their processors so that she can attach all 65,536 processors for a production run of her application.

```
(cm:detach)
NIL
```

George releases the processors he had attached so that Jane can use them.

```
(cm:global-unsigned-add 0 cm:*cube-address-length*)
(error)
```

George tries to take the sum again, just to see what will happen. An error is signalled, indicating that he no longer has hardware attached.

4 Using The Connection Machine System From *Lisp

In *Lisp only one function call is necessary to both allocate processors and initialize them for use. The *Lisp function `*cold-boot` resets the Connection Machine hardware and internal *Lisp state. It is the very first thing to do before executing any *Lisp code. Additionally `*cold-boot` calls `cm:attach` to allocate processors if there are none already attached (and by default gets the smallest available unit of processors). The `*cold-boot` function is also used with the simulator, but the processors are simulated on the front end.

The `*cold-boot` function takes an optional keyword argument `:initial-dimensions` which specifies the number of processors to allocate. When the number of processors is greater than the number of available processors, the virtual processor mechanism will re-configure the machine using virtual processors to simulate the requested dimensions. If `*cold-boot` is called without the optional `initial-dimensions` argument, it will default to the same value as the previous call to `*cold-boot`. If there is no previous value, it will default to the current physical hardware size.

It is also possible to explicitly specify what hardware configuration to use by calling the PARIS instruction `cm:attach` before calling `*cold-boot`, as explained in the previous section.

Here are some examples of `*cold-boot`. First, assume a 16K system with no processors attached to any front end. Executing

```
(*cold-boot)
```

would allocate an 8K machine, configured as in a 64×128 processor configuration. Now specify initial dimensions of 128×128 . Executing

```
(*cold-boot :initial-dimensions '(128 128))
```

would allocate an 8K machine, configured as a 128×128 , with 2×1 virtual processors being allocated on each physical processor. A 16K machine requires no virtual processors to reach 128×128 .

```
(cm:attach :16k)
```

```
(*cold-boot :initial-dimensions '(128 128))
```

5 Basic Connection Machine Operation

The full Connection Machine system consists of a Connection Machine computer and up to four front-end processors. These front-end processors are connected to the Connection Machine hardware over a length of ribbon cable that comes out of the back of the Connection Machine cabinet and connects to an interface board in the front end.

Near the front center of the machine is an Emergency Power Off Switch. Pressing the Emergency Power Off Switch turns off power to the entire machine, and is intended for emergencies only. The switch trips several circuit breakers in the machine, and a Thinking Machines Field Engineer must be called to reset these breakers before the machine can be restarted.

Directly above the Emergency Power Off Switch on the front of the Connection Machine is a control panel which is used both for turning on and off the power to 16K units of the machine and for displaying the status of connections between front end processors and the Connection Machine hardware. Figure 2 shows the control panel and emergency power switch. This figure shows the control panel used on the 16K processor systems. A later figure (figure 3) shows the control panel for a 64K processor system; there are minor differences between the two.

5.1 Using The Control Panel On A 16K Machine

Only one power switch, α , is used on a 16K processor system. That switch controls the power to the front left 16K processors. When the switch is turned clockwise to the vertical position, power is turned on, and an indicator light above the switch turns on to indicate that the power is on.

On a 16K processor system, the machine can be used as either a 16K processor system or as two 8K processor systems. The Front End Status Displays α and β show which front

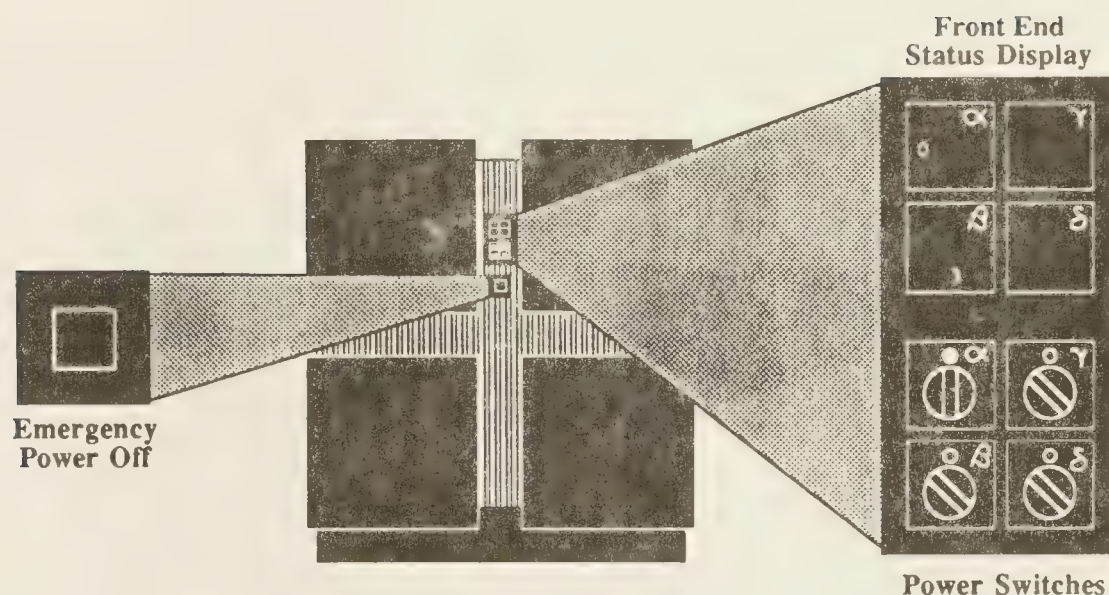


Figure 2: Front Panel for 16,384 Processor Machine

end system is connected to which section of the machine, α showing which front end the upper left 8K is connected to and β showing which section of the machine the bottom left 8K is connected to. These connections are displayed as single digits between 0 and 3, corresponding to each of the 4 front end systems that can be attached to the Connection Machine system.

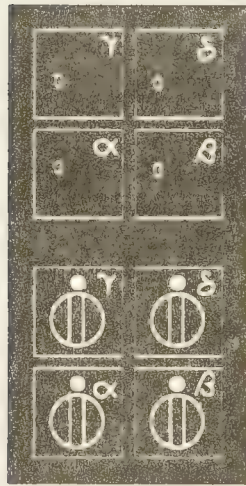
The Control Panel in 2 shows a 16K machine, with power turned on. The top 8K section of the machine, α , is connected to front end number 0, and the bottom 8K section, β , is connected to front end number 3.

This scheme extends easily. On a machine with 32K processors, power switch γ turns on the front right 16K processors, and Front End Status Displays γ and δ show which front end system is connected to the top and bottom left quarters.

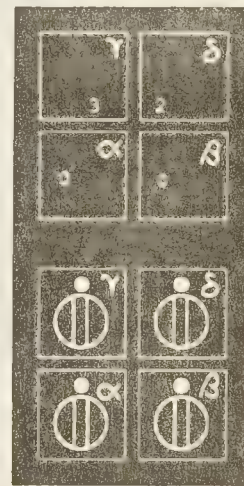
5.2 Using The Control Panel On A 64K Machine

On a 64K processor system there is a slightly different control panel with slightly different operating instructions. The major visual difference is that the Greek letters on both the power switches and the status display are not in the same place they are on the 16K control panel. The power switches each control one 16K segment of the machine: α the front left, β the front right, γ the back left, and δ the back right. The status displays refer to the same

processor arrangement, and show which front end is connected to which 16K segment. On the full 64K machine, the Connection Machine processors are allocated in groups of 16K; it is not possible to allocate an 8K unit.



Panel A



Panel B

Figure 3: Front Panel for 65,536 Processor Machine

In figure 3, Panel A shows a full 64K Connection Machine system attached to one single front end, identified as "0". Panel B shows a 64K system with a 32K segment attached to front end 0, and two 16K segments, one attached to front end 3 and one attached to front end 2.

Connection Machine segments are allocated to front end processors in power of two increments. On the 64K machine, a front end processor can be connected to an 16K, 32K, or 64K unit of the machine. There are rules that govern which segments of the machine can be combined together to make these units—for example, not any two 16K units can be combined to make a 32K unit. The two 16K processor segments α and β can be combined to make a 32K machine as can segments γ and δ . All other combinations are not supported.

6 Connection Machine Programming

Although there is nothing inherently difficult about writing a program for a data level parallel computer, it is an experience few have had. Being able to look at a problem and see its natural parallel divisions comes with experience. Thinking Machines Technical Report 86.14, *Introduction to Data Level Parallelism*, is a good introduction to solving problems on a massively parallel computer. The document has several case studies of application programs that have been written for the Connection Machine system, and is a rich source book for someone new to data level parallelism. A copy of that report is included with the system documentation. Another good reference is a book by W. Daniel Hillis, *The Connection Machine* (MIT Press, Cambridge, Massachusetts, 1985), which goes into the design issues and philosophies that led to building the Connection Machine computer.

There are two Connection Machine programming languages that run on the Symbolics 3600 family of machines: *Lisp and PARIS. *Lisp is a high level language, implemented as a set of extensions to Common Lisp. PARIS is a low level language, and is analagous to the assembly language used on serial machines. For almost all applications, *Lisp is the better language to use. Its benefits are similar to those of higher level languages on serial machines:

- Less Connection Machine hardware background is needed to write programs
- More work can be done with fewer instructions

PARIS is primarily used when there is a section of a program that must run at the fastest possible speed. Program inner loops written in PARIS are easily combined with *Lisp programs. As with high level languages on serial machines, focus first on writing a clear, functioning program, and then focus on increasing its speed.

7 Diagnostics

There are two types of whole system diagnostics that run on the Connection Machine hardware: one quick test that does a quick verification of the hardware, and one complete test that thoroughly exercises the machine. These diagnostics should be run any time a hardware problem is suspected. Warning: running these diagnostics destroys the contents of the memory in the attached Connection Machine processors.

To execute the quick diagnostics run the function:

```
(cm:hardware-test-fast)
```

This test executes in less than a minute, and writes its results to the screen. This is what will appear on the screen if no errors are detected:

```

Data: 1 2
Identity:
Select:
WELO/WEHI:
ALU:
Flags:
Cond:
NEWS: Bypassing NEWS for PINTB
Cube
T

```

One pass of the longer form of the diagnostic, `cm:hardware-test-complete`, takes several hours to run (a 16K processor machine executes the diagnostics faster than a 64K machine). This diagnostic should be run whenever the Connection Machine hardware is going to be idle for a significant amount of time, and at a minimum should be run every evening. With these diagnostics, even intermittent problems can be quickly detected. This form of the diagnostics runs continuously until aborted by the user, and is invoked by:

```
(cm:hardware-test-complete)
```

The function writes its output to both the 3600 screen and to a file. The name of the file is `local:>cm>log>3600-name.log`.

8 User Support

The most effective way to ask a question or report a software problem is to send mail to the ARPANET address `bug-connection-machine@think`. This mailing list is monitored by a number of our technical people, and you will get a reply to your message within 48 hours Monday to Friday, be it an answer to the question or a note simply saying when we will be able to get you an answer.

If you encounter an error that halts your work, including malfunctioning hardware and software that has a glaring unworkable error, you should call our headquarters at (617)876-1111 and ask for User Support. We prefer to get electronic mail, since more descriptive information can be included with the message.

Error reports are easily sent by typing `control-m` to the 3600 debugger, which sets up a mail window including a trace of what was happening at the time the error occurred. Be sure to verify that mail is being sent to the right location, since in many cases the debugger will send mail to `bug-lispm` and not to `bug-connection-machine`. When sending a software problem report, be sure to include enough of the code that caused the problem so that we can reproduce the error.

Connection Machine System Software

Release Notes

Version 4.0

**Thinking Machines Corporation
Cambridge, Massachusetts**

First printing, June 1987

The information in this document is subject to change without notice and should not be construed as a commitment by Thinking Machines Corporation. Thinking Machines Corporation reserves the right to make changes to any products described herein to improve functioning or design. Although the information in this document has been reviewed and is believed to be reliable, Thinking Machines Corporation does not assume responsibility or liability for any errors that may appear in this document. Thinking Machines Corporation does not assume any liability arising from the application or use of any information or product described herein.

Connection Machine is a registered trademark of Thinking Machines Corporation.

Paris and *Lisp are trademarks of Thinking Machines Corporation.

Symbolics, Symbolics 3600, and Genera are trademarks of Symbolics, Inc.

VAX and ULTRIX are trademarks of Digital Equipment Corporation.

UNIX is a trademark of AT&T Bell Laboratories.

Copyright © 1987 by Thinking Machines Corporation. All rights reserved.

Thinking Machines Corporation
245 First Street
Cambridge, MA 02142-1214
(617) 876-1111

Contents

1. System Software Components	1
1.1 Releases Superseded	1
2. Major Features	2
3. The VAX ULTRIX Front End	4
3.1 Restrictions to VAX ULTRIX Front End Support	4
3.2 Restrictions to the Lisp Environment on VAX ULTRIX	4
4. The Symbolics Lisp Machine Front End	6
4.1 Enhancements	6
4.2 Restrictions	6
5. Paris: The C Interface	7
6. Paris: The Lisp Interface	8
6.1 Corrections	8
6.2 Restrictions	9

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, the record of a backtrace or other error-tracing operation, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142-1214
-------------------	--

Internet Electronic Mail:	customer-support@think.com
--------------------------------------	----------------------------

Usenet Electronic Mail:	ihnp4!think!customer-support
------------------------------------	------------------------------

Telephone:	(617) 876-1111
-------------------	----------------

For Symbolics users only:

The Symbolics Lisp machine, when connected to the Internet network, provides a special mail facility for automatic reporting of Connection Machine system errors. When such an error occurs, simply press CTRL-M to create a report. In the mail window that appears, the To: field should be addressed as follows:

To: `bug-connection-machine@think.com`

Please supplement the automatic report with any further pertinent information.

1. System Software Components

Beginning with Version 4.0, Connection Machine System Software unifies the releases of several system software components:

- Software support for front-end computers
- Interfaces to Paris, the Connection Machine Parallel Instruction Set, from higher level languages
- Connection Machine standard languages

With this release and in the future, all Connection Machine (CM) software supporting the above components is upgraded simultaneously and released under the same version number.

Releases of optional CM products—additional programming languages and applications—will be independent of releases of CM System Software. Each optional CM product will have its own version number, although the timing of its release will normally coincide with a release of Connection Machine System Software.

1.1 Releases Superseded

As a result of the unification of Connection Machine System Software, Version 4.0 supersedes the following earlier releases of CM Software:

- Connection Machine Software Release 11 (November 1986), which contains support for the Symbolics Lisp machine front end
- Paris Release 2.7 (July 1986), which is an interface to Paris from Lisp
- *Lisp Release 1.7 (July 1986), a CM standard language

Version 4.0 also contains major new components—additional front-end support and a second interface to Paris—as outlined in Section 2 of these *Release Notes*.

2. Major Features

Connection Machine System Software Version 4.0 supports two types of front-end computer, provides two interfaces to Paris, and provides one CM standard language.

The front-end options for the Connection Machine are:

- A Symbolics 3600-series Lisp machine running Symbolics Genera 7.
- A Digital Equipment Corporation VAX (certain 8000-series models) running ULTRIX Version 2. ULTRIX is Digital's implementation of a UNIX system.

The available interfaces to Paris are:

- Paris callable from Lisp or from any Lisp-based CM language.
Paris/Lisp can be used on either front-end computer from within a Common Lisp environment. Symbolics Genera 7 includes a Common Lisp environment; a VAX ULTRIX system must have Lucid Common Lisp installed.
- Paris callable from C.
Paris/C can be used only on the VAX ULTRIX front end.

The available CM standard language is *Lisp. *Lisp can be used on either front-end computer from within a Common Lisp environment.

The remainder of these *Release Notes* contains information that updates the current documentation of the two front ends and the two Paris interfaces. Updates of the current documentation of *Lisp appear in a separate set of release notes, **Lisp Release Notes*, Version 4.0, June 1987.

Section 3. The VAX ULTRIX Front End

This section contains minor updates of the information presented in *Connection Machine User's Guide: Using a VAX ULTRIX System as a Front End*, Version 4.0, June 1987.

Section 4. The Symbolics Lisp Machine Front End

This section contains updates of the information presented in *Connection Machine User's Guide: Using the Symbolics 3600 as a Front End*, June, 1987.

Section 5. Paris: The C Interface

This section contains minor updates of the information presented in *Connection Machine Parallel Instruction Set (Paris): The C Interface*, Version 4.0, June 1987.

Section 6. Paris: The Lisp Interface

This section contains updates of the information presented in *Connection Machine Parallel Instruction Set (Paris): The Lisp Interface*, Release 2.7, July 1986.

3. The VAX ULTRIX Front End

CM System Software Version 4.0 is the first release to support VAX computers running the ULTRIX system as a front end to the Connection Machine. This component is documented in *Connection Machine User's Guide: Using a VAX ULTRIX System as a Front End*, June 1987. The information in this section updates that document.

3.1 Restrictions to VAX ULTRIX Front End Support

- The diagnostic tests `cm:hardware-test-fast` and `cm:hardware-test-complete` can be run on either front end, but only from within a Lisp environment. (See *Connection Machine User's Guide* for the front end you wish to use for information on running diagnostics.)
- On the VAX model 8350 only, certain tests of the suite `cm:hardware-test-complete` are shortened to reduce the amount of time this test suite takes to run. This suite is abbreviated neither on the Lisp machine front end nor on any other VAX models.

3.2 Restrictions to the Lisp Environment on VAX ULTRIX

- The Lisp environment on the VAX ULTRIX front end must be Lucid Common Lisp; no other vendor's implementation of Common Lisp is supported by CM System Software.
- ULTRIX has a text segment limit of 6 megabytes. This limit may directly affect your ability to save working images that include CM system software. For use where such an image exceeds the limit, a program called `trimtext` is provided. The program alters the Lisp image by placing part of the text into the data segment, which reduces the text segment size and allows the image to be run.

For example, from within the Lisp environment you would save an image by typing:

```
> (system:disk-save "my-cmworld" :full-gc t)
```

If the image exceeds the text segment limit, the following error will appear when you try to invoke the image from ULTRIX:

```
% my-cmworld
Not enough core
```


The `ULTRIX size` command shows the sizes of the text and data segments. If the problem is caused by a text segment of size greater than 6 megabytes, and the data segment is not too large, to the `ULTRIX` system type:

```
% trimtext my-cmworld
```

If both the text and data segments are too large, you will not be able to invoke the image.

- Connection Machine Software on the VAX `ULTRIX` front end does not support the full functionality of Lisp available on the Lisp machine front end. Specifically:
 - Double-precision floating-point numbers may not be used in Lisp. This is a limitation of Lucid Common Lisp. (Double-precision floating-point numbers may be used in C.)
 - The flavors interface provided by Lucid Common Lisp is not included in the Lisp worlds provided with this CM software release.
- When the Lucid Common Lisp environment is invoked from within a directory that is mounted on a remote file server, a message is printed before the Lisp environment is entered. This message does not appear when the Lisp environment is invoked from within a directory that is mounted on the local file system. However, it has no effect on the Lisp environment's functionality. The following example supposes the directory `/u7` is mounted on a remote file server and shows the message:

```
% pwd
/u7/jones
% lisp
>>Error: Illegal argument to SYSCALL
```

```
Debugger must find new anchor for stack... Found anchor.
SYSCALL:
```

```
->
```

4. The Symbolics Lisp Machine Front End

CM System Software Version 4.0 is the third release of software supporting the Symbolics Lisp machine as a front end to the Connection Machine. This component is documented in *Connection Machine User's Guide: Using the Symbolics 3600 as a Front End*, June 1987. The information in this section updates that document.

4.1 Enhancements

- The `show-herald` command now shows what version of the CM software is loaded.
- It is now possible to load CM software in the hardware configuration into a Lisp machine that is not physically connected to a CM. This allows world loads that include CM software to be built on Lisp machines that are not connected to a CM.

4.2 Restrictions

- The Lisp machine front end must be running Symbolics Genera 7; Connection Machine System Software Version 4.0 is not backward compatible with Symbolics Version 6.1.
- All programs created under Release 11 must be re-compiled under Version 4.0. The two releases are not binary compatible.

5. Paris: The C Interface

Connection Machine System Software Version 4.0 is the first release to support the C interface to the CM Parallel Instruction Set (Paris/C). This component is documented in *Connection Machine Parallel Instruction Set (Paris): The C Interface*, Version 4.0, June 1987.

The following are the known restrictions to the functionality of Paris/C.

- Only the following virtual processor ratios are allowed, on either front end:

1×1	2×1	4×1	8×1
	2×2	4×2	8×2
		4×4	8×4
			8×8

Each physical processor has 4K bits of memory. The Paris instructions requiring larger amounts of stack space might not have enough memory for virtual processor ratios of 4×4 (16 virtual processors per physical processor) and higher.

- The following functions incorrectly set the overflow flag and produce incorrect results when a field contains the most negative value possible:

<code>CM_round</code>	<code>CM_ceiling</code>
<code>CM_truncate</code>	<code>CM_floor</code>

- The function `CM_f_move_decoded_constant` produces incorrect results.
- The following functions produce infinity as a result if an operation results in an overflow, and they do not properly handle infinity as a source value:

<code>cm:f+</code>	<code>cm:f*</code>
<code>cm:f-</code>	<code>cm:f/</code>

6. Paris: The Lisp Interface

Connection Machine System Software Version 4.0 is the second release of the Lisp interface to the Connection Machine Parallel Instruction Set (Paris/Lisp). This component is documented in *Connection Machine Parallel Instruction Set (Paris): The Lisp Interface*, Release 2.7, July 1986. The information in this section updates that document.

6.1 Corrections

- The functions `cm:global-min` and `cm:global-max` did not return the correct results in the case where no processors were selected. This problem has been corrected.
- The functions `cm:float`, `cm:float-new-size`, `cm:f+`, and `cm:f-` now set the overflow flag properly.
- The function `cm:f/` now clears the test flag properly.
- The function `cm:float-sqrt` now sets the test flag for the case that returns `-0`, and properly handles infinity as a source value.
- The following functions were previously executed in unselected processors if the test flag was set. This problem has been corrected.

`cm:max-constant`
`cm:min-constant`

`cm:unsigned-max-constant`
`cm:unsigned-min-constant`

- Previously, the functions `cm:float-negate` and `cm:float-abs` behaved as though all processors were selected in the case where the destination and source fields were identical. This problem has been corrected.
- Previously, the following functions incorrectly signaled an error when their source and destination fields were identical. This problem has been corrected.

`cm:float-sqrt`

`cm:unsigned-mod`
`cm:unsigned-rem`

- Previously, the function `cm:f-` did not signal an error when the source and destination fields overlapped, and it produced incorrect results when the fields were identical. Both problems have been corrected.

- The following functions now signal an error when their source and destination fields overlap in any way:

<code>cm:integer-length</code>	<code>cm:unsigned-integer-length</code>
<code>cm:logcount</code>	<code>cm:unsigned-logcount</code>
<code>cm:shift</code>	<code>cm:unsigned-shift</code>
<code>cm:gray-code-from-integer</code>	

- Previously, the functions `cm:integer-length` and `cm:unsigned-integer-length` produced incorrect results for a certain range of numbers. This problem has been corrected.
- Previously, the function `cm:shift` incorrectly set the overflow flag for a shift of 0. Also, this function did not shift the destination at all when the source field contained the most negative number possible. Both problems have been corrected.
- Previously, the functions `cm:isqrt` and `cm:unsigned-isqrt` did not produce correct results with integers of length 128. This problem has been corrected.
- Previously, the function `cm:unsigned-random` was neither generating random numbers properly, nor producing correct results with a limit argument greater than 2^{16} . These problems have been corrected.
- The Paris/Lisp simulator now supports integers greater than 63 bits in length.
- Previously, in the Paris/Lisp simulator only, the function `cm:unsigned-floor` produced negative floating-point values. This problem has been corrected.

6.2 Restrictions

- Only the following virtual processor ratios are allowed, on either front end:

1×1	2×1	4×1	8×1
	2×2	4×2	8×2
		4×4	8×4
			8×8

Each physical processor has 4K bits of memory. The Paris instructions requiring larger amounts of stack space might not have enough memory for virtual processor ratios of 4×4 (16 virtual processors per physical processor) and higher.

- The following functions incorrectly set the overflow flag and produce incorrect results when a field contains the most negative value possible:

`cm:round`
`cm:truncate`

`cm:ceiling`
`cm:floor`

- The following functions produce infinity as a result if an operation results in an overflow, and they do not properly handle infinity as a source value:

`cm:f+`
`cm:f-`

`cm:f*`
`cm:f/`

- The function `cm:float-move-decoded-constant` produces incorrect results.

COMPANY PROPRIETARY
THINKING MACHINES CORPORATION

Thinking Machines Corporation

The Essential *LISP Manual
Release 1, Revision 7

July 1986

© 1986 Thinking Machines Corporation
All Rights Reserved

This notice is intended as a precaution against inadvertent publication and does not constitute an admission or acknowledgement that publication has occurred or constitute a waiver of confidentiality. The information and concepts described in this document are the proprietary and confidential property of Thinking Machines Corporation.

Document number 1-0003-1-7

© 1986 Thinking Machines Corporation.

"Connection Machine" is a registered trademark of Thinking Machines Corporation.

"*LISP" and "PARIS" are trademarks of Thinking Machines Corporation.

"Symbolics 3600" and "Zetalisp" are trademarks of Symbolics, Inc.

"VAX" is a trademark of Digital Equipment Corporation.

This document corresponds to release 1, revision 7
of the *LISP programming language

The information in this document is subject to change without notice and should not be construed as a commitment by Thinking Machines Corporation. Thinking Machines Corporation reserves the right to make changes to any products described herein to improve functioning or design. Although the information in this document has been reviewed and is believed to be reliable, Thinking Machines Corporation does not assume responsibility or liability for any errors that may appear in this document. Thinking Machines Corporation does not assume any liability arising from the application or use of any information or product described herein.

Acknowledgments

*LISP is a language designed to run on the Connection Machine system, Thinking Machines Corporation's data parallel computer. The design and implementation of the language, as well as the creation and revision of this manual, are the result of the efforts of many people at Thinking Machines. Cliff Lasser and Steve Omohundro are the primary designers and authors of the *LISP language and this manual. Some of the other "Thinkers" who were involved in one or more ways are: Guy Blelloch, Dave Christman, Danny Hillis, Brewster Kahle, Janet MacLaren, JP Massar, Bruce Nemnich, Charles Perkins, George Robertson, John Rose, Jim Salem, and Guy Steele.

Contents

1	Introduction	1
1.1	The Use of Different Typefaces in This Manual	2
2	Connection Machine Computers and *LISP	3
2.1	The Connection Machine Computer Organization	3
2.2	*LISP Terminology	5
2.3	Overview of ESSENTIAL *LISP	6
2.3.1	Pvars	6
2.3.2	Selection	8
2.3.3	*Defun	9
2.3.4	Communication	10
2.3.5	The Format of Function Definitions	10
2.4	Configuration Constants and Functions	10
2.4.1	The Size of the Machine	11
2.4.2	Pre-defined Pvars	11
3	The Pvar Data Structure	12
3.1	Creating New Pvars	12
3.2	Allocating Local Pvars	14
3.3	Setting the Value of Pvars	14
3.4	Reading and Writing Fields in Specific Processors	15
4	Processor Selection	16
5	!! Functions	19
5.1	Predicate !! Functions	19
5.2	Logical !! Functions	20
5.2.1	Logical !! Operations on Numbers	20
5.2.2	Logical !! Operators	21
5.3	Numerical !! functions	22
5.4	Miscellaneous !! Functions	24
5.5	User !! and * Functions	26
5.6	Debugging Tools	27

6	Parallel Global Memory References	28
6.1	Parallel Global Memory References	28
6.2	Processor Addressing	32
7	Global Operations	34
8	Actually Using the Hardware	35
9	Actually Using the Simulator	37
10	Potentially Troublesome Situations	38
10.1	Pvar Values in Non-selected Processors	38
10.2	The Extent of Pvars	38
10.3	Using Mapcar on Functions that Return Pvars	39
10.4	*Warm-boot	39
10.5	*Cold-boot	39
10.6	Pref!!, Pref-grid!! and Pref-grid-relative!!	40
10.7	Using Pref!! and Friends Efficiently	40
10.8	Multiple Values	41
A	Some Example Programs	42
B	List of ESSENTIAL *LISP Commands	50

Chapter 1

Introduction

Welcome to *LISP!! This language (pronounced *star lisp*), which runs on Connection Machine[®] data parallel computers, is an extension of COMMON LISP. This ESSENTIAL *LISP manual describes the fundamental instruction set used in *LISP. More advanced features are described in the COMPLETE *LISP manual.

This manual is organized into the following chapters:

1. Introduction

This chapter. It provides an overview of the ESSENTIAL *LISP manual's organization, and explains the significance of the different typefaces used throughout the manual.

2. Connection Machine Computers and *Lisp

This chapter provides an overview of the Connection Machine computer and of the language ESSENTIAL *LISP, including:

- a description of the conceptual structure and naming conventions of the language
- example expressions that suggest the feel of the language
- a description of the conventions used in presenting language functions
- an introduction to the Lisp constants that specify the configuration parameters of the Connection Machine system

3. The Pvar Data Structure

Pointers to Connection Machine memory are stored in Lisp objects called pvars. This chapter defines the data structure known as a *pvar* and the operations that create pvars and act on them.

4. Processor Selection

This chapter describes the mechanisms for selecting the set of processors that will perform an operation.

5. !! Functions

In *LISP, the names of instructions that return parallel variables (*pvars*) as their values end with !! (pronounced "bang-bang"). This suffix is meant to look like two parallel lines and

indicates that a pvar is returned. This chapter describes the functions for combining and comparing pvars numerically and logically, and describes the mechanisms for defining new functions of this type.

6. Parallel Global Memory References

This chapter discusses the mechanisms for moving data between Connection Machine processors.

7. Global Operations

This chapter describes operations for combining data globally from all Connection Machine processors.

8. Actually Using the Hardware

This chapter describes the process of actually using the hardware, including procedures for logging in to and resetting the Connection Machine system.

9. Actually Using the Simulator

This chapter describes the process of actually using the *LISP simulator.

10. Potentially Troublesome Situations

This chapter describes potential ambiguities and trouble spots in the ESSENTIAL *LISP language definition.

11. Appendix A: Some Example Programs

This chapter provides some sample programs written in *LISP.

12. Appendix B: List of ESSENTIAL *LISP Commands

This chapter summarizes all the commands described in ESSENTIAL *LISP and can be used as a quick reference guide. Commands are grouped by functionality, rather than by name; so if users know what they want to do, but not which *LISP command to use in order to do it, then this is the place to go for help.

1.1 The Use of Different Typefaces in This Manual

Throughout the ESSENTIAL *LISP manual, actual code (including actual names of functions) will generally appear in the following typeface:

`(cons a b)`

Names that stand for pieces of code (metavariables) will generally be written in *italics*. Note that in function descriptions, the names of the parameters will appear in italics for expository purposes.

Chapter 2

Connection Machine Computers and *LISP

2.1 The Connection Machine Computer Organization

A Connection Machine data parallel computer consists of a large number of simple processors. Each has some associated local memory and is integrated into a highly connected communications network. A typical configuration might have 65,536 processors with 4,096 bits of memory each. Typical applications utilize data structures that have components spanning many Connection Machine processors. The goal of *LISP is to provide a language for creating and manipulating these structures in parallel.

*LISP is an extension of COMMON LISP. COMMON LISP provides a rich set of functions for manipulating data structures in a serial computer (see *Common Lisp* by Guy L. Steele Jr. [Steele 84]). *LISP provides versions of these functions that execute in many Connection Machine processors in parallel.

*LISP has several important properties:

- *LISP programs are easy to write and debug. *LISP introduces a minimal set of new concepts to ordinary serial computer programming.
- As with other machines, the complexity of programming usually lies entirely in the complexity of the actual application. Many applications are easier to implement in *LISP due to the raw processing power of the Connection Machine system!
- *LISP is built entirely on top of COMMON LISP. This allows the user to mix ordinary Lisp code with *LISP code.
- Most *LISP language features map directly into Connection Machine hardware capabilities; therefore, users quickly develop an intuition for predicting program performance.
- *LISP includes primitives for interfacing efficiently to low-level utilities such as PARIS (see *Connection Machine Parallel Instruction Set (PARIS)* [Thinking Machines Corporation 86a]) and other Connection Machine languages.

The *LISP language is still evolving, and Connection Machine programming is still in its infancy. Only aspects of the language that are well understood and fairly certain to not change have been doc-

umented here, hence the name ESSENTIAL *LISP. Many more functions and features are documented in the COMPLETE *LISP manual [Lasser in preparation].

This manual attempts only to explain the *LISP language. For a general overview of the Connection Machine system and how to program it, see *The Connection Machine* by W. Daniel Hillis [Hillis 85] and *Introduction to Data Level Parallelism* [Thinking Machines Corporation 86b].

Here are the primary concepts of the *LISP language:

- *LISP programs execute in a front-end computer, typically a Symbolics Lisp machine or a DEC VAX. As a side effect of running the *LISP program, the front-end computer generates instructions (at the PARIS level) for the Connection Machine processors to execute. Every so often, the front-end computer will transfer data or results of computations to or from the Connection Machine.
- *LISP programs refer to memory in the Connection Machine processors through Lisp objects called "Pvars" (Parallel variables). *LISP provides a number of mechanisms for managing Connection Machine memory through pvars. These objects contain information about the actual location of the memory in the Connection Machine and the possible types of values stored in that memory. A pvar looks like a large vector of COMMON LISP values, and each of these values are stored one per Connection Machine processor. Values stored in the Connection Machine may be integers, floating point numbers, booleans, or any other Lisp object (for example: 0, 102, -5, 10.333, t, nil, hi-there). As with COMMON LISP, the ESSENTIAL *LISP programmer need not be concerned with type coercion, since it is done automatically.
- *LISP programs control the set of Connection Machine processors that are executing instructions. This set may range from all processors in the machine to no processors.

Given these concepts, a *LISP program typically consists of these parts:

- Permanent Connection Machine storage declarations.
- Connection Machine static data structure creation code. This often involves substantial transfers of data from the front-end computer to the Connection Machine.
- Main body of *LISP program. This in turn typically contains:
 - Temporary Connection Machine storage allocation.
 - Selection of Connection Machine processors for expression evaluation.
 - Parallel expression evaluation with the result stored in a destination.
 - Massive communication of data inside the Connection Machine.
 - Transfer of results back to the front-end computer.

2.2 *LISP Terminology

The following are descriptions of terms that are used with a specific meaning in *LISP:

- **Processors**

The conceptual entities that operate on data in parallel are called *processors*. Often these correspond to actual hardware processors, but sometimes a single hardware processor simulates several conceptual processors. In this case, the simulated processors are referred to as *virtual processors*. This simulation is transparent to the programmer. The Connection Machine Parallel Instruction Set manual describes in detail the implications and mechanisms of virtual processors.

- **Cube Address**

Each processor has a unique *cube address*. The cube addresses of the processors in the Connection Machine system range between zero and the number of processors less 1. On a 65,536 processor machine, this would be between 0 and 65,535, though this may be much larger with the use of virtual processors.

- **Grid Address**

A specific processor can also be identified by one or more numbers, referred to as the processor's *grid address*. The number of coordinates in a *grid address* is determined by the number of dimensions the Connection Machine system is simulating. For example, one might refer to a processor with a grid address of (3,4,1) in a three-dimensional machine. A two-dimensional machine representing a two-dimensional grid would require two grid address coordinates. Note that the numbering scheme for grid addresses in a one-dimensional machine is *not* necessarily the same as that for cube addresses. ESSENTIAL *LISP requires the user to choose a machine configuration at initialization time and to keep that configuration throughout the program. The COMPLETE *LISP manual describes how to dynamically change the size and shape of the Connection Machine system.

The current version of *LISP supports configuring the processors using grid addressing in *only* two-dimensional configurations. This restriction will be removed in a future implementation of the language.

- **Field**

The most primitive form of data in *LISP is a *field*. A field is a string of contiguous bits in the same memory locations of *each* processor. A field in a processor may contain any valid Lisp value. The different processors of a field may contain different types of values as well. A field exists only inside the Connection Machine processors. A field is somewhat analagous to a COMMON LISP vector.

- **Pvar**

A Lisp object that contains all the information necessary to manage a field is called a *pvar*. This is short for *parallel variable*.

- **Contents of a Pvar**

The phrase *contents of a pvar* is often used to refer to the values stored in the field in the Connection Machine memory that is described by the pvar.

- **Component of a Pvar**

A component of a pvar refers to the contents of a pvar in a single Connection Machine processor.

- **Currently Selected Set**

Most *LISP operations are only carried out in a subset of the Connection Machine processors. This subset is called the *currently selected set* and is specified by using *LISP special forms, such as *all, *when, *cond, and *if.

- **!!**

The names of instructions that return pvars as their values end with !! (pronounced “bang-bang”). This suffix is meant to look like two parallel lines and indicates that a parallel variable is returned. It is an excellent idea for user-defined functions to obey this convention (though nothing enforces it), because it helps ensure that pvars are produced only in contexts where they can be used. It is an error to produce pvars in contexts where they cannot be used (see Chapter 10).

There are a few ESSENTIAL *LISP forms whose names do not end in !!, such as *when, *all and *let, that, nonetheless, may optionally return a pvar.

- *****

All ESSENTIAL *LISP functions that do parallel computation and do not end in !! begin with * (pronounced “star”). Note that in COMMON LISP * is used as a prefix operator to denote multiplication, but that it is used in *LISP to denote parallel operations (hence, the name *LISP!).

- **Parallel Equivalent Of**

This phrase is used to describe a *LISP function with reference to a COMMON LISP function; for example, “Mod!! is the parallel equivalent of the COMMON LISP function mod.” This means that mod!! performs the same calculation as mod, only mod!! performs the operation in parallel using each component of each pvar argument.

For a complete list of ESSENTIAL *LISP commands, see Appendix B.

2.3 Overview of ESSENTIAL *LISP

This section contains some example expressions to give the reader an idea of what *LISP expressions actually look like. Precise definitions of all the functions used occur in other sections of the manual.

2.3.1 Pvars

The following create five sample pvars:

```
(*defvar a)
(*defvar b (!! 5) "This is a documentation string.")
(*defvar c (!! -2.67))
(*defvar d t!!)
(*defvar e (1+!! (self-address!!)))
```


The last four have been initialized with specific values: **b** is a Lisp symbol that contains a pvar containing the integer 5 in each processor; **c** contains the floating point number -2.67 in each processor; **d** contains the Lisp symbol T in each processor; and **e** contains an integer that is the cube address of the next higher processor. See Section 3.1 for a more detailed description of ***defvar**.

The function **pref** can be used to read out some of the above values. The arguments of **pref** are a pvar and a cube address. This is analogous to the COMMON LISP **aref**; the pvar is equivalent to an array and the cube address to the array index.

For example,

```
(pref c 0)
```

returns the Lisp value -2.67, since that is what is contained in pvar **c** in processor 0.

Similarly,

```
(pref d 365)
```

returns the Lisp value T because that is what is contained in pvar **d** in processor 365. See Section 3.4 for a more detailed description of **pref**.

*LISP uses the COMMON LISP macro **setf** to turn accessor expressions into modifier expressions. For example, there is no function that does the opposite of **pref**. To write into a single processor of a pvar, one would write something like:

```
(setf (pref b 0) 15)
```

The form **(pref b 0)** would now return 15 because 15 was just stored in pvar **b** in processor 0.

The following demonstrate arithmetic operations on the example pvars:

```
(*set a (+!! b c))
```

will make the contents of pvar **a** be the sum of the contents of pvar **b** and pvar **c**. Notice that because **c** contains floating point values, the integers contained in **b** are properly coerced to floating point and the result in **a** will be floating point as well. See Section 3.3 for a more detailed discussion of ***set**.

Expressions can be nested:

```
(*set a (-!! b (*!! a (!! 2))))
```

This sets **a** to the difference of **b** and 2 times **a**. This simple expression causes thousands of operations to go on simultaneously! See Section 5.3 for a more detailed discussion of numerical **!!** functions.

2.3.2 Selection

When the Connection Machine is initialized, each and every processor will be in a state to execute all *LISP instructions in parallel. However, it may be necessary to execute a sequence of instructions in some subset of all Connection Machine processors.

One way of temporarily subselecting some Connection Machine processors is to wrap the *when macro around a body of forms. For example, to select the set of all processors whose cube addresses (contained in the pvar returned by the function `self-address!!`) end in 1, one might do the following:

```
;; Create a pvar that is True in all odd processors
(*defvar odd-address-p (==!! (!! 1) (mod!! (self-address!!) (!! 2))))

;; Now select all processors with odd cube addresses and do ...
(*when odd-address-p!! ...)
```

In another case, it may be desirable to do an operation in the processors that have both an even cube address, and in which the pvar `a` contains zero. Two natural ways to do this are to (1) use the logic functions to select the correct set:

```
(*when (and!! (not!! odd-address-p)
              (==!! a (!! 0)))
  (*set a (+!! a b)))
```

or equivalently, (2) to nest *when expressions:

```
(*when (not!! odd-address-p)
  (*when (==!! a (!! 0))
    (*set a (+!! a b))))
```

It might also be advantageous to do an operation with a temporary variable allocated. For example, if a programmer wants to do an operation in all processors whose addresses are divisible by four, he or she might say:

```
(*let ((g (mod!! (self-address!!) (!! 4))))
  (*when (==!! g (!! 0))
    (*set a (+!! a b))))
```

This first creates a temporary variable `g` and loads it up with the two lowest order bits of the (`self-address!!`). In all processors in which this is 0, the *set operation is performed.

To perform different operations in processors whose addresses have a remainder of 0, 1, 2 or 3 after dividing by 4, the following might be used:

```
(*let ((g (mod!! (self-address!!) (!! 4))))
  (*cond
    ((==!! g (!! 0)) (*set a (+!! a b)))
    ((==!! g (!! 1)) (*set a (-!! a b)))
    ((==!! g (!! 2)) (*set a (*!! a b)))
    ((==!! g (!! 3)) (*set a (/!! a b)))))
```

There are also *LISP expressions analogous to the Lisp if:

```
(*if (<!! z x)
      (*set y (!! 5))
      (*set y (!! 6)))
```

and

```
(*set y
  (if!! (<!! z x)
        (!! 5)
        (!! 6)))
```

Note that `cond!!` and `if!!` return a pvar that must be stored into some destination, whereas `*cond` and `*if` are executed only for side effect.

See Chapter 4 for more detailed descriptions of `*when`, `*cond` and `*if`; Section 6.2 for the definition of `self-address!!`; Section 3.2 for a discussion of `*let`; Sections 5.2.1 and 5.2.2 for definitions of the logical functions; and Section 5.4 for a description of `if!!`.

2.3.3 *Defun

To define functions that can take pvars as arguments or return them as values, use `*defun` instead of `defun`. To define a function that takes two pvar arguments and returns their sum, difference, product, or quotient (depending on whether the processor's address has remainder 0, 1, 2 or 3 when divided by 4 in all processors in the currently selected set), use something like the following:

```
(*defun four-function!! (pvar-a pvar-b)
  (*let ((address-bits (mod!! (self-address!!) (!! 4)))
        (answer))
    (*cond
      ((=!! address-bits (!! 0)) (*set answer (+!! pvar-a pvar-b)))
      ((=!! address-bits (!! 1)) (*set answer (-!! pvar-a pvar-b)))
      ((=!! address-bits (!! 2)) (*set answer (*!! pvar-a pvar-b)))
      ((=!! address-bits (!! 3)) (*set answer (/!! pvar-a pvar-b))))
    answer))
```

This may now be used like any other `!!` function, as in:

```
(*set a (four-function!! (+!! a (!! 4)) (-!! a b)))
```

To pass a *LISP function as an argument, use `*funcall`. For example:

```
(defun *compose (*f *g x)
  (*funcall *f (*funcall *g x)))

(*set a (*compose 'sqrt!! '1+!! (!! 8)))
```

acts like:

```
(*set a (sqrt!! (1+!! (!! 8))))
```

See Section 5.5 for the precise definitions of both `*defun` and `*funcall`.

2.3.4 Communication

This section demonstrates how to cause the processors to communicate with one another.

One connectivity pattern that can be specified upon initialization is a two-dimensional grid in which each processor has a neighbor on the north, east, west, and south (or NEWS for short). It is possible to sum the value contained in **a** in the four neighbors of each processor, and store the result back in **a** as follows:

```
(*set a (+!! (pref-grid-relative!! a (!! 0) (!! 1))      ;north
          (pref-grid-relative!! a (!! 1) (!! 0))      ;east
          (pref-grid-relative!! a (!! -1) (!! 0))      ;west
          (pref-grid-relative!! a (!! 0) (!! -1))      ;south
          ))
```

(The effect of indexing out of bounds is ignored for the time being.)

To have the first 100 processors in the Connection Machine write the contents of field **b** into the field **a** of those processors whose addresses are 7 larger, do:

```
(*when (<!! (self-address!!) (!! 100))      ;select first 100 processors
      (setf (pref!! a (+!! (self-address!!) (!! 7))) b))
```

Note that **pref!!** is the parallel version of **pref**; all selected processors will perform in parallel a **pref** from the processor of their choice.

Finally, to find the single maximum value of **a** in all even processors, one possibility is to do:

```
(*when (=!! (!! 0) (mod!! (self-address!!) (!! 2)))
      (*max a))
```

2.3.5 The Format of Function Definitions

The format of function definitions in this manual strive to be as compatible as possible with the format used in the COMMON LISP manual [Steele 84]. Argument names can restrict the type of an argument; argument names that end in the suffix "pvar" must be pvars. The name "integer-pvar" restricts the argument to a pvar whose fields in the currently selected set of processors must all contain integers.

2.4 Configuration Constants and Functions

*LISP makes it convenient to simulate in software a configuration of processors that is different from their physical configuration. It is important to write software that can take advantage of this flexibility. In addition, it is desirable to write software that will run on machines with differing amounts of physical hardware.

The variables defined in Section 2.4.1 specify the parameters of the machine as perceived by the user's program. If a program uses only these constants and functions, it will run in any configuration.

The size of simulated configuration of processors is specified through the ***cold-boot** function (see Chapter 8).

2.4.1 The Size of the Machine

The user must not modify any of the following variables. They are set by `*cold-boot`. See Chapter 8 on page 35 for more information on `*cold-boot`.

`*number-of-processors-limit*` [Variable]

This variable specifies the effective number of processors a user program sees. For a machine with 65,536 physical processors, each simulating 16 processors, this variable will contain 1,048,576.

`*log-number-of-processors-limit*` [Variable]

This variable provides the logarithm, base 2, of the number of processors available. This will be a floating point number if `*number-of-processors-limit*` is not a power of 2.

`*number-of-dimensions*` [Variable]

This variable is defined when `*cold-boot` is run. Its value is the number of dimensions given. `*Cold-boot` defaults this variable to be 2. The current hardware implementation supports only two dimensions.

`*current-cm-configuration*` [Variable]

This variable is a list containing each dimension's size in the current configuration.

`dimension-size dimension`

This function returns one more than the maximum allowable grid address for the specified *dimension*. Note that *dimension* is zero based; for example, in a two-dimensional machine, the first dimension is dimension zero and the second is dimension one.

2.4.2 Pre-defined Pvars

`t!!` [Constant]

This is a pvar whose contents in each processor is the Lisp symbol T.

`nil!!` [Constant]

This is a pvar whose contents in each processor is the Lisp symbol NIL.

Chapter 3

The Pvar Data Structure

The basic abstraction in *LISP is the pvar. A pvar is a Lisp object that references a field of memory in the Connection Machine system. It contains everything necessary to describe the field. In ESSENTIAL *LISP, the contents of pvars may be any valid Lisp object. As in COMMON LISP, coercion between data types and allocation of memory is handled automatically.

This portion of the manual is organized as follows:

- Section 3.1 describes `*defvar`, `allocate!!`, and `*deallocate`, which create new pvars and destroy old ones.
- Section 3.2 describes `*let` and `*let*`, which allocate temporary pvars.
- Section 3.3 describes `*set`, which allows each processor to set the value contained in the field defined by a pvar.
- Section 3.4 describes functions for explicitly reading and writing Lisp values from or to specific processors of a pvar.

3.1 Creating New Pvars

To create a permanent named pvar, use `*defvar` (analogous to the Lisp `defvar`). To create a permanent, unnamed pvar, use `allocate!!`; and to create a temporary, named pvar, use `*let` (analogous to the Lisp `let`).

`*defvar` *symbol* &optional *pvar documentation-string*

This creates a new pvar that is permanently allocated. *symbol* will contain the allocated pvar. The optional argument *pvar* may be any pvar or pvar expression. `*defvar` will create a new pvar of the type and size of the given *pvar*, initialize it to the given pvar's contents, and `setq` the *symbol* to that new pvar. If no *pvar* argument is given, the *symbol* will contain a pvar whose values are undefined. Note that `*cold-boot` will reset the values of all pvars allocated by `*defvar`. This form returns *symbol*.

Some example uses of `*defvar` are:

```
(*defvar a)
(*defvar b (!! 5))
(*defvar c (+!! b (!! 6)))
(*defvar d t!!)
(*defvar e (self-address!!))
(*defvar f c)
```

**deallocate-*defvars &rest pvar-names*

This function will permanently flush the specified *pvar-names*. If *pvar-names* is *nil* or *:prompt*, the user will be prompted for each pvar ever declared with **defvar*. If *pvar-names* is *:all*, then all **defvars* will be deleted. Before deleting all **defvars*, users should be certain that none of the library functions they call depend on any pvars created with **defvar*.

Here are some sample uses:

```
(*deallocate-*defvars 'foo)           ;delete foo pvar
(*deallocate-*defvars 'foo 'bar)       ;delete foo and bar pvars
(*deallocate-*defvars)                 ;prompt user for pvars to delete
(*deallocate-*defvars :prompt)         ;prompt user for pvars to delete
(*deallocate-*defvars :all)            ;deletes all *defvars
```

allocate!! &optional pvar

Like **defvar*, this creates a permanent pvar, except that the created pvar is simply returned. It is up to the user to store it someplace. *allocate!!* is very useful for making Lisp arrays or structures that contain permanent pvars. If no *pvar* is specified, then the returned pvar will have values which are undefined; otherwise, *pvar* is used to initialize the newly allocated pvar.

Some example uses are:

Assume *c* is a vector:

```
(dotimes (j (length c))
  (setf (aref c j) (allocate!!!)))
```

and

```
(setq a (allocate!!!))
(setq b (allocate!! (!! 5)))
(setf (aref c 4) (allocate!! t!!!))
```

**deallocate pvar*

This deallocates the given pvar if it was permanently allocated (i.e., it was defined using either *allocate!!* or **defvar*). It is an error to use a pvar after it has been deallocated. The order in which pvars are deallocated does not matter. This function returns no values.

pvarp object

This returns *T* if the argument is a pvar and *NIL* if it is not.

3.2 Allocating Local Pvars

*LISP maintains a stack of temporary pvars for its own purposes. When a `!!` function returns a pvar, it has been allocated on this stack. *LISP's ability to arbitrarily nest `!!` expressions stems from its maintenance of this stack. While this automatic allocation takes care of many situations, there are times when it is desirable to explicitly allocate a temporary variable. The ESSENTIAL *LISP functions for doing this are `*let` and `*let*`:

`*let` (*{(symbol &optional pvar)}**) &rest body [Macro]

The first expression following the `*let` should be a list of lists, each specifying one temporary pvar. The elements of each sublist should consist of a Lisp symbol whose value will be the temporary pvar, followed by an optional pvar or pvar expression that will be copied into the new one. This format is very similar to that of `*defvar`, the only difference being that several pvars can be created at once; these pvars only survive for the extent of the form. It is an error to try to refer to these pvars outside of the body of the `*let`. In other words, the *symbols* have lexical scope (as in COMMON LISP), whereas the pvars themselves have dynamic extent that terminates when the `*let` form is exited. (For a more detailed discussion of this, see Chapter 10 on page 38 of this manual.) `*let` returns the value of the last form of the body, regardless of whether that value is a pvar. It is legitimate to return a temporarily bound pvar. `*let` is *not* able to return multiple values.

`*let*` (*{(symbol &optional pvar)}**) &rest body [Macro]

This function behaves in the same manner as `*let` except that, as in COMMON LISP, the defining expressions are evaluated in sequence, so that previous bindings affect the evaluation of future initialization forms.

Example `*let*` expressions might look like:

```
(*let* ((a)
        (b (!! 8))
        (c (*!! b (!! 528)))
        (d (!! -2.715))
        (e (self-address!!)))
  (some-pvar-function a b c d e) ;This may modify a,b,c,d and/or e
  (+!! a b c d e)                ;This returns a pvar

;; take the global maximum of bits 16-31 of the self pointer
(*let ((a (load-byte!! (self-address!!) (!! 16) (!! 16))))
  (*max a))
```

3.3 Setting the Value of Pvars

The `*set` special form allows the contents of one pvar to be set to the contents of another. The field is set in those processors that are currently selected. A `*set` expression returns no value. `*set` takes multiple pairs, which are set sequentially. `*set` is sometimes used in conjunction with `*all` to set the contents of one pvar to the contents of another in *all* processors, not just the selected ones.

***set {*pvar-1* *pvar-2*}* [Macro]**

This sets the contents of *pvar-1* to the contents of *pvar-2* in all processors of the currently selected set. Note that both of the arguments are evaluated.

Some examples of the use of this function are:

```
(*set a (+!! b c))
```

```
(*all (*set a (!! -1) b a c (!! -3)))
```

```
(*when (>!! d (!! 4)) (*set a b))
```

3.4 Reading and Writing Fields in Specific Processors

This section describes functions for getting data into and out of the Connection Machine processors. They are independent of the currently selected set and return, as Lisp values, the read or written Lisp value. (The current functions read or write a single value at a time. Functions for efficiently moving blocks of data will be available in the future.)

pref *pvar* *address*

This function returns, as a Lisp value, the contents of the field specified by *pvar* in the processor whose cube address is *address*. **setf** may be used with **pref** to write a value into a single processor of a *pvar*.

```
(pref foo 17)
```

returns the contents of *pvar* *foo* from processor 17.

```
(setf (pref foo 17) (* 19 89))
```

sets the contents of *pvar* *foo* for processor 17 to 1691.

pref-grid *pvar* &rest *addresses*

This function returns, as a Lisp value, the contents of the field specified by *pvar* in the processor whose grid address is given by *addresses*. There must be as many addresses as there are dimensions (as specified with ***cold-boot**). **setf** may be used with **pref-grid** to write a value into a single processor of a *pvar*.

```
(pref-grid bar 4 7)
```

returns the contents of *pvar* *bar* from processor (4,7) (on a two-dimensionally configured machine).

```
(setf (pref-grid bar 4 7 8) (* 19 89))
```

sets the contents of *pvar* *bar* for processor (4,7,8) (on a three-dimensionally configured machine) to 1691.

Chapter 4

Processor Selection

Most operations are only executed in a subset of Connection Machine processors known as the *currently selected set*. Some of the special forms in ESSENTIAL *LISP that change the currently selected set are **all*, **when*, **cond*, and **if*. These special forms select processors based on the result of a *pvar* expression. Any processor in which the *pvar* expression evaluates to NIL is eliminated from the selected set.

Although these macros may modify the currently selected set, they all obey the discipline of restoring the currently selected set to its previous state upon completion. Also, they may be nested as deeply as desired.

It is common for user functions to have a **all* surrounding their bodies to ensure that they are starting out with the complete machine selected. Using the functions described in this section, the selected set is whittled down to select only the processors that should do a given operation. The body of these forms is *always* executed, even if there are no selected processors.

Note: In the current implementation, of the forms below that return values, none are configured to allow the return of multiple values. It is an error to attempt to return multiple values from any of these forms.

**all &rest body* [Macro]

This form selects all processors. Its body is executed with the currently selected set equal to the entire machine. The value of the final expression in the body is returned whether it is a Lisp value or a *pvar*.

**when pvar &rest body* [Macro]

This form *subselects* from the currently selected set. Thus every processor that is unselected when **when* is called remains unselected in the body of the **when*. It selects processors in which *pvar* is non-NIL. **Even if there are no selected processors, ALL forms in the body are evaluated.** The value of the final expression in the body is returned whether it is a Lisp value or a *pvar*.

**if pvar then-form &optional else-form* [Macro]

This form is analogous to the Lisp *if*. *then-form* is performed in all processors of the currently selected set in which *pvar* is not NIL. The optional *else-form* is evaluated in all other processors of

the currently selected set in which the *pvar* is NIL. Even if there are no selected processors, both *then-form* and *else-form* are evaluated. Unlike Lisp's *if*, this function returns no values and is executed only for its side effects (see *if!!* in Section 5.4).

***cond {(pvar {form}*)}* [Macro]**

This form is analogous to the Lisp *cond*. Unlike the Lisp *cond*, **cond* evaluates all clauses; however, the currently selected set is determined by the *pvar* expressions. The *n*th consequent is evaluated with a selected set made up of initially selected processors that didn't pass the first *n* - 1 tests, but did pass the *n*th one. *t!!* selects all remaining processors in the initial selected set. Even if there are no selected processors, all consequent forms are evaluated. Unlike Lisp's *cond*, this function returns no values and is executed only for its side effects (see *cond!!* in Section 5.4).

with-css-saved {form}* [Macro]

This form is used whenever control flow would abnormally pass out of a *LISP form that restricts the currently selected set (e.g. using *throw*, *return-from*, or *go*, to leave the body of a **when*). *with-css-saved* uses an *unwind-protect* to trap these events and force the currently selected set back to its state at the time the *with-css-saved* form was begun. *with-css-saved* returns what is returned by the evaluation of the last form of its body.

do-for-selected-processors (symbol) &rest body [Macro]

This form evaluates *body* as many times as there are active processors, each time with *symbol* bound to the cube address of a different active processor. Like COMMON LISP's *dotimes*, the *return* statement may be used to exit the *do-for-selected-processors* form immediately. Normally, *do-for-selected-processors* returns NIL.

Some examples of the use of these functions are:

```
(*all (*set a b))

(*when (=!! a b) (*set e (+!! c d)))

(*cond ((=!! a (! 1)) (*set e (+!! b c)))
        ((not!! (=!! c d)) (*set f (*!! b c)))
        (t!! (*set f (! 9))))

(*if (=!! c d) (*set e f) (*set g h))

(*set a (=!! b c))
(*when a (*set b (-!! b)))
```

```
(*defun f (x y)
  "Returns y divided by x for y greater than 0.  Returns NIL if any x is 0."
  (block foo
    (with-css-saved
      (*when (>!! y (!! 0))
        (*if (==!! (!! 0) x)
          (return-from foo nil)
          (/!! y x)
        )))))
```

Chapter 5

!! Functions

This section introduces a variety of functions that work within an individual processor and that return a pvar. Recall that in *LISP, it is conventional for functions that return a pvar to end in !!.

This portion of the manual describes the following:

- the predicates that return Boolean pvars (Section 5.1)
- the logical functions (Sections 5.2.1 and 5.2.2)
- the numerical functions (Section 5.3)
- the forms that allow the user to define his or her own !! functions (Section 5.4)

5.1 Predicate !! Functions

The functions in this section are typically used as predicates within **when* expressions. They return a pvar that contains T in all processors of the currently selected set in which the predicate holds, and a NIL in those in which it does not.

=!! numeric-pvar &rest numeric-pvars

This returns a pvar that contains T in each processor where the argument pvars contain equal values and NIL elsewhere. To see if a pvar is equal to a Lisp constant, use an expression like:

(=!! foo (!! 5))

If only one argument pvar is given, the returned pvar will be t!!.

/=!! numeric-pvar &rest numeric-pvars

This returns a pvar that contains T in each processor where the argument pvars contain unequal values and NIL elsewhere. If only one argument is given, the returned pvar will be t!!. (Note: On Symbolics Lisp machines, when using the Zetalisp reader, the symbol “/” is a reader macro character, so one must use “//=!!”.)

<!! *numeric-pvar &rest numeric-pvars*

This returns a pvar that contains T in each processor where the argument pvars contain values which are in strictly increasing order and NIL elsewhere. If only one argument pvar is given, the returned pvar will be t!!.

>!! *numeric-pvar &rest numeric-pvars*

This returns a pvar that contains T in each processor where the argument pvars contain values which are in strictly decreasing order and NIL elsewhere. If only one argument pvar is given, the returned pvar will be t!!.

<=!! *numeric-pvar &rest numeric-pvars*

This returns a pvar that contains T in each processor where the argument pvars contain values which are in non-decreasing order and NIL elsewhere. If only one argument pvar is given, the returned pvar will be t!!.

>=!! *numeric-pvar &rest numeric-pvars*

This returns a pvar that contains T in each processor where the argument pvars contain values which are in non-increasing order and NIL elsewhere. If only one argument pvar is given, the returned pvar will be t!!.

5.2 Logical !! Functions

5.2.1 Logical !! Operations on Numbers

This section contains a variety of Boolean functions that operate bitwise on the bits of the fields described by the argument pvars and return a pvar that holds the result. These functions may be used only on pvars whose contents are integers. The returned pvar will usually contain positive integers.

lognot!! *integer-pvar*

This returns a pvar whose bits are the logical complement of the bits in *integer-pvar*.

logior!! &rest *integer-pvars*

This returns a pvar whose bits are the logical inclusive or of the bits in *integer-pvars*. If there are no pvars, then (!! 0) is returned.

logxor!! &rest *integer-pvars*

This is the parallel equivalent of the COMMON LISP function *logxor*. If there are no pvars, then (!! 0) is returned.

logand!! &rest integer-pvars

This returns a pvar whose bits are the logical and of the bits in *integer-pvars*. If no *pvars* are given, then (**!! -1**) is returned.

logeqv!! &rest integer-pvars

This is the parallel equivalent of the COMMON LISP function *logeqv*. If no *pvars* are given, then (**!! -1**) is returned.

5.2.2 Logical !! Operators

*LISP provides several logical operators. Some of these operators (**and!!** and **or!!**) are special because they temporarily subselect the currently selected set as they evaluate their arguments. The rest of the operators are normal !! functions.

As in COMMON LISP, a value is true if it is anything other than NIL.

not!! pvar

This returns T for all processors in which *pvar* is NIL, and NIL otherwise.

and!! &rest pvars

[Macro]

This evaluates the *pvars* from left to right in all selected processors. As soon as one of the *pvars* evaluates to NIL in a processor, that processor is removed from the currently selected set for the remainder of the **and!!**. **and!!** will return the value of the last pvar for all selected processors in which all the *pvars* are true and NIL otherwise. If no *pvars* are given, then **t!!** is returned.

or!! &rest pvars

[Macro]

This evaluates the *pvars* from left to right in all selected processors. As soon as one of the *pvars* evaluates to non-NIL in a processor, that processor is removed from the currently selected set for the remainder of the **or!!**. The value returned for each processor will be the first pvar that evaluated to non-NIL. If none of the *pvars* are true, then NIL is returned. If no *pvars* are given, then **nil!!** is returned.

xor!! &rest pvars

This performs the xor function on all the *pvars*. If no *pvars* are given, then **nil!!** is returned. In each processor this returns T if there are an odd number of arguments that are true and otherwise returns NIL.

eql!! pvar1 pvar2

This is the parallel equivalent of the COMMON LISP function *eql*.

eq!! *pvar1 pvar2*

This is the parallel equivalent of the COMMON LISP function **eq**.

integerp!! *pvar*

This is the parallel equivalent of the COMMON LISP function **integerp**.

floatp!! *pvar*

This is the parallel equivalent of the COMMON LISP function **floatp**.

numberp!! *pvar*

This is the parallel equivalent of the COMMON LISP function **numberp**.

zerop!! *numeric-pvar*

This is the parallel equivalent of the COMMON LISP function **zerop**.

5.3 Numerical !! functions

This section describes the elementary numerical functions. As with COMMON LISP, the results of these functions are always numerically correct. For example, the result of an addition is never truncated, no matter how much memory is required to represent the result. If not enough memory is available, an error will be signalled.

These functions each return results of the same type as the most expensive of their arguments (e.g. if all arguments are integers, the result will generally be an integer; but if any arguments are floats, the result will be a float).

!! *lisp-expression*

This returns a pvar containing the result of *lisp-expression* in each processor.

+!! &rest *numeric-pvars*

This adds the contents of the argument pvars. If there are no arguments, then (!! 0) is returned.

-!! *numeric-pvar* &rest *numeric-pvars*

This subtracts the contents of the second through last argument's pvars from the contents of the first. If there is only one argument, the result is that argument negated.

!! &rest *numeric-pvars

This multiplies the contents of the argument pvars. If there are no arguments, then (!! 1) is returned.

/!! numeric-pvar &rest numeric-pvars

This returns the quotient of *pvar* by the rest of the *pvars*. If there is only one argument, the result is the inverse of *pvar*. Note: */!!* presently always returns a *pvar* whose contents are all floating point numbers. If there is only one argument, it is an error if that argument has any field whose value is 0. If there is more than one argument, it is an error if any argument but the first has any field whose value is 0.

1+!! numeric-pvar

This increments the argument *pvar* by 1. A new *pvar* is returned.

1-!! numeric-pvar

This decrements the argument *pvar* by 1. A new *pvar* is returned.

min!! numeric-pvar &rest numeric-pvars

This returns a *pvar* that is the minimum of all the argument *pvars*.

max!! numeric-pvar &rest numeric-pvars

This returns a *pvar* that is the maximum of all the argument *pvars*.

mod!! numeric-pvar integer-pvar

This is the parallel equivalent of the COMMON LISP function *mod*. It is an error if *integer-pvar* contains zero in any processor.

ash!! integer-pvar count-pvar

This is the parallel equivalent of the COMMON LISP function *ash*. *Count-pvar* may be an integer or float *pvar*.

truncate!! numeric-pvar &optional divisor-numeric-pvar

This is the parallel equivalent of the COMMON LISP function *truncate*, except that only one value (the first) is computed and returned.

round!! numeric-pvar &optional divisor-numeric-pvar

This is the parallel equivalent of the COMMON LISP function *round*, except that only one value (the first) is computed and returned.

ceiling!! numeric-pvar &optional divisor-numeric-pvar

This is the parallel equivalent of the COMMON LISP function *ceiling*, except that only one value (the first) is computed and returned.

floor!! *numeric-pvar &optional divisor-numeric-pvar*

This is the parallel equivalent of the COMMON LISP function `floor`, except that only one value (the first) is computed and returned.

sqrt!! *non-negative-pvar*

This returns the non-negative square root of the given *pvar*.

isqrt!! *non-negative-integer-pvar*

This is the parallel equivalent of the COMMON LISP function `isqrt`.

random!! *limit-pvar*

This returns a *pvar* whose contents is a random value between 0 inclusive and *limit-pvar* exclusive for each processor.

5.4 Miscellaneous !! Functions

load-byte!! *from-pvar position-pvar size-pvar*

This function returns a *pvar* whose contents are positive integers. It consists of bits extracted from *from-pvar* starting at bit position *position-pvar*, where 0 represents the least significant bit. In any processor in which zero bits are extracted, the resulting field contains zero. This operation is especially fast when both *position-pvar* and *size-pvar* are constants, as in `(!! lisp-value)`. *from-pvar* must be a *pvar* containing integers, while *position-pvar* and *size-pvar* must be *pvars* containing non-negative integers. Out of range bits are treated as zero for positive integers (for example, `(load-byte!! (!! 1) (!! 2) (!! 3))` returns a *pvar* that contains zero in each processor), and one for negative integers (for example, `(load-byte!! (!! -1) (!! 2) (!! 3))` returns a *pvar* that contains 7 in each processor).

deposit-byte!! *into-pvar position-pvar size-pvar byte-pvar*

This returns a *pvar* whose contents are a copy of *into-pvar* with the low order *size-pvar* bits of *byte-pvar* inserted into the bits starting at location *position-pvar*.

When the *into-pvar* is positive (negative), zeros (ones) are appended as high order bits of *byte-pvar* as needed. The returned value may have more bits than *into-pvar* if the inserted field extends beyond the most significant bit of *into-pvar*. For example, `(deposit-byte!! (!! 3) (!! 1) (!! 2) (!! 2))` will return `(!! 5)`. This function is especially fast when both *position-pvar* and *size-pvar* are constants, as in `(!! lisp-value)`. *Into-pvar* and *byte-pvar* must contain integers, while *position-pvar* and *size-pvar* must be *pvars* containing non-negative integers only.

if!! *pvar then-pvar else-pvar*

[Macro]

This returns a *pvar* that contains the contents of the *then-pvar* in all processors in which *pvar* is non-NIL, and the contents of *else-pvar* in all processors in which *pvar* is NIL. For the execution of the *then-pvar* expression, the currently selected set is set to all processors that passed the predicate, whereas for the execution of the *else-pvar* the currently selected set is set to all the selected processors that failed the predicate. (See also **if*, which is executed only for side effect.)

This is equivalent to:

```
(*let ((result)
      (temp-pred pvar))
  (*when temp-pred
    (*set result then-pvar))
  (*when (not!! temp-pred)
    (*set result else-pvar))
  result
)
```

An example that demonstrates the usefulness of *if!!* is the following function to take the absolute value:

```
(*defun abs!! (pvar)
  (if!! (>!! pvar (!! 0)) pvar (-!! pvar)))
```

cond!! *{(pvar {form}*)}*

[Macro]

If there are no clauses, *cond!!* returns *nil!!*. Otherwise, *cond!!* is roughly equivalent to the following pseudo-code:

```
(if!! pvar-1
  (progn all-the-forms-for-clause1)
  (cond!! (rest clauses)))
```

However, if there are no forms for a given clause, the *pvar* itself is used as the value of the clause, analogous to the COMMON LISP *cond*. (See also **cond*, which is executed only for side effect).

enumerate!!

This returns a *pvar* that contains a unique number in each selected processor from 0 up to one less than the number of selected processors. The numbers are ordered, 0 being put in the processor with the smallest cube address, 1 being put in the processor with the next smallest cube address, etc. If all processors in the Connection Machine are selected, this is equivalent to the function *self-address!!*.

rank!! numeric-pvar predicate

Rank!! returns a pvar containing the values 0 through 1 less than the active number of processors, such that for all values *v1* and *v2*, and for all active processors *p1* and *p2*, if *v1* < *v2*, then the value of *numeric-pvar* in processor *p1* satisfies the serial analog of *predicate* with respect to the value of *numeric-pvar* in processor *p2*. (Currently, *predicate* must be the symbol <=!!, and its serial analog is the predicate <=.) If there are no active processors, **rank!!** returns *numeric-pvar*.

5.5 User !! and * Functions

This section provides information necessary for defining and using new !! functions.

Pvar arguments are passed by *reference*, not by *value*. Thus the contents of pvars passed as arguments can be changed using ***set**. It is generally considered poor form for a function to modify one of its arguments; instead, most *LISP functions return a new pvar whose contents may be a modified copy of one of the arguments.

***defun**

[Macro]

This is analogous to the COMMON LISP **defun** and must be used in place of it in defining all user functions that might take as an argument a pvar or that might return a pvar as a result. This returns, as a symbol, the name of the function being defined. Like the COMMON LISP **defun**, the body may contain declarations and a documentation string.

If, in a given file, a function *foo* defined by ***defun** is called before it is defined textually in the file, or is called but is not defined in the current file, then the user must declare that *foo* is actually a function defined by ***defun** and is not a regular function defined by **defun**. One makes such a declaration with the COMMON LISP function **proclaim**. For example:

```
(proclaim '(*defun foo))
```

Failure to make such declarations will result in incorrectly compiled code.

***funcall function &rest arguments**

[Macro]

This is used just like COMMON LISP's **funcall**, but with functions defined with ***defun**. One may not **funcall** a ***defun**'ed function.

***apply function arg &optional more-args**

[Macro]

This is used just like COMMON LISP's **apply**, but with functions defined with ***defun**.

5.6 Debugging Tools

pretty-print-pvar *pvar &key mode format per-line start end*

This prints out the contents of a pvar in all processors. The default format used is “~S ” (This function calls the COMMON LISP function `format`. See the COMMON LISP manual for more information on format directives.) If *per-line* is not given, no newlines are ever printed between values; otherwise, *per-line* values are printed out and then a newline is output. *Mode* can either be `:cube` (the default) or `:grid`, in which case the pvar is printed out using grid addressing rather than cube addressing. If *start* and/or *end* are given, these restrict the range of processors over which values are printed out. Since this function is so useful, an alias, `ppp`, is also defined. `pretty-print-pvar` returns no values.

If `pretty-print-pvar` accesses a processor that has no defined value for *pvar*, then the symbol `*` is printed out.

list-of-active-processors

This simply returns a list of cube addresses of all the currently selected processors. The order of this list is not specified. Since this function is so useful, an alias, `loap`, is also defined. This could be written as:

```
(defun list-of-active-processors ()
  (let ((return-list nil))
    (do-for-selected-processors (processor)
      (push processor return-list))
    return-list))
```

pretty-print-pvar-in-currently-selected-set *pvar &key format start end*

This function prints out the the cube address and value of *pvar* for all processors in the currently selected set. Since this function is so useful, an alias, `ppp-css`, is also defined. *format* defaults to “~S ”. This function returns no values.

Chapter 6

Parallel Global Memory References

6.1 Parallel Global Memory References

This section describes the mechanisms for moving data between the Connection Machine processors in parallel. The high-speed Connection Machine router network provides global memory references from many processors in parallel. The function that does communication is `pref!!`, the parallel version of `pref`.

`pref!! pvar-expression cube-address-pvar` [Macro]

`pref!!` will return a `pvar` that contains the value of `pvar-expression` from the processors addressed by `cube-address-pvar`. This function evaluates `pvar-expression` differently from other *LISP operators; instead of evaluating the `pvar-expression` in the currently selected set, it is evaluated in the context of the processor from which the data is being retrieved. Unlike the `pvar-expression`, `cube-address-pvar` is evaluated normally (i.e., in the processors of the currently selected set).

If the value of `pvar-expression` in a single processor is being accessed by more than one other processor, the Connection Machine system arranges for all those other processors to get the same value.

`pref-grid!! pvar-expression &rest grid-address-pvars &key border-pvar` [Macro]

`pref-grid!!` will return a `pvar` that contains the value of `pvar-expression` from the processors addressed by `grid-address-pvars`. This function evaluates `pvar-expression` differently from other *LISP operators; instead of evaluating the `pvar-expression` in the currently selected set, it is evaluated in the context of the processor from which the data is being retrieved.

There must be as many `grid-address-pvars` as there are Connection Machine dimensions. Unlike the `pvar-expression`, the `grid-address-pvars` are evaluated like normal expressions (i.e., in the processors of the currently selected set).

It is an error to read from a non-existent processor. However, if `border-pvar` is provided, and if the `grid-address-pvars` in a given processor `p` access a non-existent processor, then the value of `border-pvar` in processor `p` is returned instead. During the evaluation of `border-pvar`, the currently selected set is set to *only* those processors reading off the edge of the Connection Machine grid. In order to better understand this behavior, consider the following two pieces of code:

```
(pref-grid!! source x-address y-address :border-pvar foo)

(if!! (off-grid-border-p!! x-address y-address)
  foo
  (pref-grid!! source x-address y-address))
```

These are equivalent except the `pref-grid!!` ensures that *x-address* and *y-address* are evaluated exactly once each.

Again, if the value of *pvar-expression* in a single processor is being accessed by more than one other processor, the Connection Machine system arranges for all those other processors to get the same value.

`pref-grid-relative!! pvar-expression &rest relative-address-pvars`
`&key border-pvar`

This function behaves like `pref-grid!!`, except that relative addressing is used instead of absolute addressing. An example of the use of this function is given later in this chapter.

This function is especially fast when the *relative-address-pvars* are all constants (as in `(!! x)`).

As with the serial function `pref`, `setf` may be applied to `pref!!`, `pref-grid!!` and `pref-grid-relative!!` to write into memory instead of reading from it. In this case, *pvar-expression* will be referred to as *dest-pvar* and must be a valid destination pvar.

When using `setf` in this manner, *dest-pvar* is modified only in those processors that were accessed. Processors that were not written into will retain the previous contents of *dest-pvar*. An error is signalled if a non-existent processor is addressed. This occurs when an address is out of the bounds specified by the current Connection Machine configuration.

Unlike a normal `setf` form, using `setf` with `pref!!` and its relatives does not return the pvar that was used as the new value. Instead, `setf` in these instances returns no values.

Although the Connection Machine hardware is capable of accessing the same memory for several readers without problems, the user must instruct it on how to handle collisions when several processors are simultaneously writing to the same location. Should a processor be written into by several other processors in a single memory reference, `pref!!` and its relatives (in combination with `setf`) will signal an error. The function `*pset` allows multiple writes to combine in various ways without producing errors.

`*pset combiner value-pvar dest-pvar cube-address-pvar` [Macro]

For all selected processors, *value-pvar* will be written into *dest-pvar* of the processor addressed by *cube-address-pvar*. When more than one value is written into the same address, the *combiner* determines how the values are combined. *combiner* may be one of the following:

1. **:default** — If the same address is written twice, an error is signalled. This is the same as using **setf** with **pref!!**.
2. **:overwrite** — Only one write per address is successful. All other writes are discarded.
3. **:or** — If two or more values are written into a single processor, the final value will be the logical OR of those values.
4. **:and** — If two or more values are written into a single processor, the final value will be the logical AND of those values.
5. **:logior** — If two or more values are written into a single processor, the final value will be the bitwise OR of those values. *value-pvar* must contain integers only.
6. **:logand** — If two or more values are written into a single processor, the final value will be the bitwise AND of those values. *value-pvar* must contain integers only.
7. **:add** — If two or more values are written into a single processor, the final value will be the numerical SUM of those values.
8. **:max** — If two or more values are written into a single processor, the final value will be the numerical MAXIMUM of those values.
9. **:min** — If two or more values are written into a single processor, the final value will be the numerical MINIMUM of those values.

The **:logior** and **:logand** combiners are especially fast. The **:or** and **:and** are faster when the *pvar* being sent contains only T's and NIL's.

***pset-grid combiner value-pvar dest-pvar &rest grid-address-pvars** [Macro]

This is analogous to ***pset**, except that the grid addressing is used.

***pset-grid-relative combiner value-pvar dest-pvar &rest relative-grid-address-pvars** [Macro]

This is analogous to ***pset-grid**, except that relative grid addressing is used. This function is especially fast when the *relative-address-pvars* are all constants (as in **(!! x)**).

The following are some sample uses of **pref!!**:

```
(*set a (pref!! b (!! 100)))
```

This reads the contents of **b** from processor 100 and stores it in *pvar* **a**. Only those components of **a** which are in processors in the currently selected set are modified.

These two forms are equivalent:

```
(*all (setf (pref!! b (self-address!!)) a))

(*all (*set b a))
```

This example writes pvar *a* into pvar *b* of all processors. Processors can read from themselves just as easily as they can read from other processors.

The next example demonstrates that the *pvar-expression* of *pref!!* is evaluated in the processor from which the data is being fetched. Remember that it is an error to read or write from a non-existent processor.

```
(*all                                ;select all processors in the CM
  (*let ((a (!! 10)))                ;initialize a to 10 in all processors
    (setf (pref
```

These two forms are equivalent:

```
(*all
  (*when (>!! (self-address!!) (!! 0))
    (*set a (pref!! (self-address!!)
                    (1-!! (self-address!!))))))

(*all
  (*when (>!! (self-address!!) (!! 0))
    (*set a (1-!! (self-address!!))))
```

Note that this example demonstrates that the *pvar-expression* of *pref!!* is evaluated in the processor from which the data is being fetched. Remember that it is an error to read or write from a non-existent processor. In the above examples, the form *(*when (>!! (self-address!!) (!! 0))* prevents processor 0 from reading processor -1.

This function:

```
(*defun sum-a-pvar (pvar)
  (pref
    (*let ((the-sum-goes-here))
      (*all (*pset :add pvar the-sum-goes-here) (!! 47)))
    the-sum-goes-here)
  47)
```

returns the sum of a pvar over all the Connection Machine processors. (Processor 47 was chosen to contain the sum for demonstration purposes only.)

The following is an example of `pref-grid-relative!!`:

```
(*all
  (*set color
    (/!!
      (+!!
        (pref-grid-relative!! color (!! -1) (!! 0) :border-pvar (!! 1))
        (pref-grid-relative!! color (!! 0) (!! -1) :border-pvar (!! 1))
        (pref-grid-relative!! color (!! 0) (!! 1) :border-pvar (!! 1))
        (pref-grid-relative!! color (!! 1) (!! 0) :border-pvar (!! 1))
        color)
      (!! 5))))
```

This example causes the value of the `color` pvar in each processor to be averaged with the 4 processors to its north, east, west and south.

6.2 Processor Addressing

This section contains functions that deal with address generation and translation. When a dimension number is required, remember that it is always zero based; in other words, the first dimension is dimension 0, the second dimension is dimension 1, and so on.

`self-address!!`

This function returns a pvar that contains the cube address of each selected processor.

`self-address-grid!! dimension-pvar`

This returns a pvar that contains the grid address, in the specified dimension, of each selected processor. Each processor may specify a different dimension through *dimension-pvar*.

`grid-from-cube-address cube-address dimension`

This function takes a *cube-address* and returns the grid address for the specified *dimension*. This function executes entirely in the front-end computer.

`cube-from-grid-address address-pvar &rest address-pvars`

This function translates a grid address consisting of (possibly) several *address-pvars* into a cube address. This function executes entirely in the front-end computer.

`grid-from-cube-address!! cube-address-pvar dimension-pvar`

This function takes a *cube-address-pvar* and returns a pvar containing the grid address for the specified *dimension-pvar* for each selected processor.

cube-from-grid-address!! *address-pvar* &rest *address-pvars*

This function translates a grid address consisting of (possibly) several *address-pvars* into a cube address for each selected processor.

off-grid-border-p!! &rest *grid-address-pvars*

This returns a boolean pvar that is true if the *grid-address-pvars* specify an address that is invalid given the current dimensions, and false otherwise. It is an error for any component of *grid-address-pvar* to not be an integer.

off-grid-border-relative-p!! &rest *relative-grid-address-pvars*

This function is identical to **off-grid-border-p!!** except that the *relative-grid-address-pvars* specify relative addresses.

Chapter 7

Global Operations

The following functions reduce the contents of a *pvar* in all selected processors into a single Lisp value, which is then returned:

**logior integer-pvar*

This returns a Lisp value that is the bitwise logior of the contents of *integer-pvar* in all selected processors. This returns the Lisp value 0 if there are no selected processors.

**logand integer-pvar*

This returns a Lisp value that is the bitwise logand of the contents of *integer-pvar* in all selected processors. This returns the Lisp value -1 if there are no selected processors.

**min numeric-pvar*

This returns a Lisp value that is the minimum of the contents of *numeric-pvar* in all selected processors. This returns the Lisp value NIL if there are no selected processors.

**max numeric-pvar*

This returns a Lisp value that is the maximum of the contents of *numeric-pvar* in all selected processors. This returns the Lisp value NIL if there are no selected processors.

**or pvar*

This returns a Lisp value of T if the contents of *pvar* is non-NIL in any selected processor; otherwise, it returns NIL. If there are no selected processors, this function returns NIL. For example, to determine if there are any processors currently selected, use (**or t!!*), which returns T only if there are selected processors.

**and pvar*

This returns a Lisp value of T if the contents of *pvar* is non-NIL in every selected processor; otherwise, it returns NIL. If there are no selected processors, this function returns T.

**sum numeric-pvar*

This returns a Lisp value that is the sum of *numeric-pvar* in every selected processor. This returns the Lisp value 0 if there are no selected processors.

Chapter 8

Actually Using the Hardware

This section describes the two `*LISP` functions (`*cold-boot` and `*warm-boot`) that allow the user to actually use the hardware. It also defines the Common Lisp package in which `*LISP` lives. For instructions on how to load the `*LISP` language into your Lisp machine, please refer to the *Connection Machine User's Guide* [Thinking Machines Corporation 86c].

The `*LISP` language lives in a package called `*LISP`. To use the language, one must be in that package:

```
(in-package '*LISP)
```

or else cause that package to be used by whatever package the user is in:

```
(use-package '*LISP)
```

On Symbolics Lisp Machines, it is critical that when the `*LISP` package is created that it inherit from the Common Lisp world and not from the Zetalisp world. To this end, it is recommended that any file containing functions that are to be put in the `*LISP` package have the following package attribute in the attribute list:

```
Package: (*LISP COMMON-LISP-GLOBAL)
```

It is also strongly suggested that users write their code in Common Lisp and use a Common Lisp Listener when running their programs.

`*cold-boot &key initial-dimensions` [*Macro*]

This function initializes `*LISP` and must be called immediately after loading in the `*LISP` software. It resets the internal state of the `*LISP` system, as well as the Connection Machine hardware. All `*defvar` pvars are reallocated and their initial values are recomputed according to the order in which they were defined.

In addition, the user may specify the *initial-dimensions* of the machine. This argument is a list of dimension sizes. This affects the behavior of `*LISP` functions such as `pref-grid!!` and `pref-grid-relative!!`. The dimensions must be powers of 2. If no *initial-dimensions* are specified, then they default to the same values as in the previous call to `*cold-boot`.

The current hardware implementation of *LISP only handles 2-dimensional grids. The simulator implementation works for n-dimensional grids.

*LISP will try to attach the necessary amount of Connection Machine hardware to satisfy the user requirements. If the hardware is not large enough, or is not of the proper shape, *LISP will try running virtual processors. The only difference the user will ever notice is a degradation of performance. An error will be signalled if sufficient hardware is not available.

*cold-boot is typically called by the initialization function of the user's software. Under normal circumstances, this need only be called at the start of a session.

Users may explicitly specify what hardware configuration to use by calling the Connection Machine Parallel Instruction Set instruction `cm:attach` before calling *cold-boot. See the *Connection Machine Parallel Instruction Set* manual for more details. *cold-boot will call `cm:attach` if the hardware is not already attached.

Here are some typical calls:

```
;; this will end up running 64 x 128 processors since that is the hardware
;; configuration for an 8k machine
(*cold-boot)                                ;8k processors

;; this will end up running 2 x 1 virtual processors on an 8k machine (which
;; is what will be allocated by default in a 16k machine)
(*cold-boot :initial-dimensions '(128 128)) ;16k processors

(cm:attach :16k)
;; will run no virtual processors because 128 x 128 is the physical size of a
;; 16k machine.
(*cold-boot :initial-dimensions '(128 128)) ;16k processors
```

*warm-boot

This function must be called whenever a *LISP program is abnormally terminated for any reason. The function will reset only certain internal *LISP and Connection Machine hardware states.

It is wise to call this function at the beginning of major entry points in the user's software, since previously run and aborted code may have left the Connection Machine hardware in an inconsistent state.

Chapter 9

Actually Using the Simulator

See the discussion about packages at the beginning of the *Actually Using the Hardware* chapter.

The `*LISP` simulator strives to be an exact simulation of the `ESSENTIAL *LISP` specification on a serial machine. The functions `*cold-boot` and `*warm-boot` work in the same manner as for the hardware, except that an upper limit of 2048 processors is imposed. The simulator is more lenient than the current hardware with respect to dimensioning the machine: The simulator allows an arbitrary number of dimensions, while the hardware currently supports only 2. Warnings are issued if one tries to configure the simulator in a way that the hardware cannot deal with.

Since the simulator and hardware implementations of `*LISP` are very different, it is unfortunately necessary to recompile code when switching from one system to the other. It is not possible to load the simulator once the hardware version of `*LISP` has been loaded, and vice versa.

The features `*LISP-HARDWARE` and `*LISP-SIMULATOR` are defined when running on hardware and on the simulator respectively so that where necessary, reader-time conditionalization of code can be done.

For instructions on how to load the `*LISP` language into your Lisp Machine, please refer to the *Connection Machine User's Guide* [Thinking Machines Corporation 86c].

The following functions exist only in the `*LISP` simulator:

`display-*lisp-function-use-statistics`

This prints out the name of each `*LISP` function that has been called at least once since `*cold-boot` or `reset-*lisp-function-use-statistics` has been executed, and the number of times each function has been called. It returns `NIL`.

`reset-*lisp-function-use-statistics`

This function resets the use count statistics displayed by `display-*lisp-function-use-statistics`. It returns `NIL`.

Chapter 10

Potentially Troublesome Situations

This chapter describes potential ambiguities and trouble spots in the ESSENTIAL *LISP language definition.

10.1 Pvar Values in Non-selected Processors

It is an error to depend on the value of a pvar in a processor that was not in the currently selected set at the time the pvar was created.

For instance,

```
(*when (<!! (self-address!!) (!! 10))
  (*let ((foo (self-address!!)))
    (print (pref foo 20))
  ))
```

The ESSENTIAL *LISP language definition does not define the value printed in the above example.

10.2 The Extent of Pvars

Unlike COMMON LISP, pvars defined using *let or *let* have dynamic extent – that is, it is an error to reference the value of a pvar once the body of its defining *let or *let* has been exited.

For example:

```
(*defun will-not-work (pvar constant)
  (funcall
    (*let ((xyzy (! constant)))
      #'(lambda (x) (*sum (+!! (!! x) xyzy)))
    )
    (pref pvar 0)
  ))
```

Since the body of the `*let` defining `xyzy` has been exited at the time the lambda-defined function is actually called, the ESSENTIAL *LISP language definition makes no guarantee that the pvar `xyzy` will still contain constant anywhere.

Note that `*let` and `*let*` allow one to return a `*let`-ed pvar as the value of the `*let` and use that returned value. It is only attempting to access a `*let`-ed value inside a lexical closure that is doomed to failure.

10.3 Using Mapcar on Functions that Return Pvars

Consider the following code:

```
(let ((pvar-list nil))
  (setq pvar-list (mapcar #'!! '(1 2 3)))
  (*set pre-defined-pvar-1
    (+!! (first pvar-list) (second pvar-list)))
  (*set pre-defined-pvar-2
    (+!! (first pvar-list) (second pvar-list)))
)
```

`pvar-list` is a list of pvars that have been allocated on the ESSENTIAL *LISP stack. The ESSENTIAL *LISP language definition makes no guarantees as to how long these pvars will remain inviolate, since they are on the stack. There is absolutely no guarantee that `pre-defined-pvar-1` and `pre-defined-pvar-2` will contain the same values.

10.4 *Warm-boot

Whenever an ESSENTIAL *LISP program has an error, and the user aborts back to top level, the `*warm-boot` function *must* be called before attempting to run any ESSENTIAL *LISP code again. This is because both the currently selected set and certain internal *LISP variables will be left in an inconsistent state. `*warm-boot` resets all Connection Machine errors without modifying the contents of the Connection Machine memory. Very bizarre behavior results when this dictate is not followed.

`*Warm-boot` is intended to be used as a top-level form. Under no circumstances should it be used inside of a `*defun` form, either lexically or in such a manner that it might be executed while a `*defun` function is being evaluated.

10.5 *Cold-boot

`*Cold-boot` is intended to be used as a top-level form. Under no circumstances should it be used inside of a `*defun` form, either lexically or in such a manner that it might be executed while a `*defun` function is being evaluated.

10.6 Pref!!, Pref-grid!! and Pref-grid-relative!!

These *LISP functions (which are actually macros) are defined to evaluate their source argument(s) in the context of the set of addresses defined by evaluating their address pvar(s). Therefore, writing a function `foo` which calls one of these functions with a source argument `s` passed into `foo` may not work because `s` will have already been evaluated by the Lisp evaluator before the body of `foo` is evaluated.

The solution is to define such functions as macros. For example:

```
(*defun foo (condition source-pvar address-pvar)
  (if condition
    (pref!! source-pvar address-pvar)
    (!! 0)
  ))
```

May not work as intended and should be rendered as

```
(defmacro foo (condition source-pvar address-pvar)
  '(if ,condition
    (pref!! ,source-pvar ,address-pvar)
    (!! 0)
  ))
```

If `source-pvar` is always a symbol and not a pvar expression, then this modification is not necessary.

10.7 Using Pref!! and Friends Efficiently

The following code, which retrieves the value of a pvar in a single processor and distributes the result to all processors:

```
(*all (pref!! any-pvar (!! 0)))
```

will, in the current implementation, take an extremely long time to execute, since the fetches are serialized. It is much faster to implement this as:

```
(!! (pref any-pvar 0)).
```

10.8 Multiple Values

The current implementation does not support the return of multiple values from any *LISP form. It is error to attempt such an operation.

Appendix A

Some Example Programs

```
;;; -*- SYNTAX: common-lisp; MODE: lisp; BASE: 10; PACKAGE: *LISP-CL; MUSER: YES; Fonts:
CPTFONTB,CPTFONT -*-
```

```
;;; Cliff Lasser 2/86
```

```
;;;
;;; This file contains working examples for the Essential *Lisp manual.
;;; Although they are in Common Lisp, they do use a couple of ZetaLisp
;;; features: the LOOP macro and 3600 specific terminal control.
;;;
```

```
;;;
;;; Example # 1: Conway's life
;;;
```

```
;;;
;;; Conway's Life is a simple 2-dimensional cellular automata algorithm.
;;; Each cell in the 2-d grid is either alive or dead. A cell becomes alive
;;; if exactly three of its neighbors are alive. A cell stays alive if two
;;; or three of its neighbors are alive. In all other cases a cell becomes
;;; dead or remains dead. A cell's neighbors are the cells to its east,
;;; west, north and south, and to its NW, NE, SW and SE.
;;;
```

```

;;;
;;; This is the main entry point for the Life program. It will do all
;;; initialization, then a few cycles of Life.
;;;
(DEFUN DO-LIFE ()
  (INITIALIZE-LIFE)
  (DISPLAY-LIFE-GRID)
  (LIFE-CYCLE 100))                                ;run 100 cycles of life

;;;
;;; This declares a PVAR that exists in all processors. Each cell in the
;;; grid is either alive or dead. If the cell is alive, CELL-ALIVE-P will
;;; contain T. If it is dead, it will contain NIL.
;;;
(*DEFVAR CELL-ALIVE-P)

;;;
;;; This will initialize *Lisp and create a random pattern on the Life grid.
;;; By default, 30% of the cells will be alive
;;;
(DEFUN INITIALIZE-LIFE (&OPTIONAL (PERCENT-ON 30))
  ;; Initialize the Connection Machine system (CMS) and *Lisp. Remember
  ;; to always call this before actually doing anything on the CMS.
  (*COLD-BOOT)

  ;; now fill the machine with a random pattern
  (*SET CELL-ALIVE-P (<!! (RANDOM!! (!! 100)) (!! PERCENT-ON)))
  )

;;;
;;; This will display the Life grid on the user's terminal
;;;
(DEFUN DISPLAY-LIFE-GRID ()
  ;; clear the screen (Symbolics 3600 specific code)
  (ZL:SEND TV:SELECTED-WINDOW :CLEAR-WINDOW)
  ;; step through each location on the x-y grid
  (LOOP FOR X FROM 0 BELOW (DIMENSION-SIZE 0)
    DO
    ;; go to next line on screen.
    (FORMAT T "~%")
    (LOOP FOR Y FROM 0 BELOW (DIMENSION-SIZE 1)
      DO
      ;; if the cell is alive, print a *. Otherwise, a space

```



```

      (IF (PREF-GRID CELL-ALIVE-P X Y)
          (PRINC "*")
          (PRINC " "))))))

;;;
;;; This function will repeatedly run the Life algorithm and display the life
;;; grid.
;;;
;;;
(DEFUN LIFE-CYCLE (NUMBER-OF-CYCLES)
  (LOOP FOR CYCLE FROM 0 BELOW NUMBER-OF-CYCLES
    DO
      ;; allocate some temporary storage for counting the number of alive
      ;; neighbors for each cell in the grid.
      (*LET ((NUMBER-OF-ALIVE-NEIGHBORS (! 0)))

        ;; each cell totals up the number of alive neighbors. Be careful to
        ;; not include the cell itself in the total.
        (LOOP FOR X FROM -1 TO 1
          DO
            (LOOP FOR Y FROM -1 TO 1
              DO
                (WHEN (NOT (= 0 X Y))
                  (*WHEN (PREF-GRID-RELATIVE!! CELL-ALIVE-P (! X) (! Y))
                    ;; only those cells with the specific neighbor alive get to
                    ;; increment the count.
                    (*SET NUMBER-OF-ALIVE-NEIGHBORS
                      (+!! NUMBER-OF-ALIVE-NEIGHBORS (! 1)))))))

        ;; based on the count of alive neighbors, each cell decides whether to
        ;; become alive or dead.
        (*SET CELL-ALIVE-P (OR!! (AND!! CELL-ALIVE-P
                                          (=!! (! 2) NUMBER-OF-ALIVE-NEIGHBORS))
                                  (=!! (! 3) NUMBER-OF-ALIVE-NEIGHBORS))))

      ;; display the life grid.
      (DISPLAY-LIFE-GRID)
    ))

```

```

;;;
;;; Example # 2: Shortest path in a graph
;;;

```

```

;;;
;;; In this example, we have the CMS find the shortest path between two cities
;;; in a graph of connected cities. Each connection between cities contains
;;; the distance between the two connected cities.
;;;

```

```

;;;
;;; The data structure used is very simple: We split up the Connection
;;; Machine processors into two sets. The first set represents the cities
;;; themselves. The second set represents the connections between cities.
;;;

```

```

(*DEFVAR CITY-P NIL "true when this processor contains a city")
(*DEFVAR CONNECTION-P NIL "true when this processor contains a connection item")

(*DEFVAR CITY-NAMES NIL "This contains the name of city for each city processor")
(*DEFVAR CITY-DISTANCE-FROM-START NIL "Distance of a city from the START processor")

(*DEFVAR CONNECTED-CITY-FROM NIL "One of the cities in a connection")
(*DEFVAR CONNECTED-CITY-TO NIL "The other city in a connection")
(*DEFVAR CONNECTION-DISTANCE NIL "The distance between the connected cities")

```

```

;;;
;;; This defines the graph of connected cities
;;;
(DEFPARAMETER LIST-OF-ALL-CITIES '(NEW-YORK LOS-ANGELES BOSTON
                                   WASHINGTON SAN-FRANCISCO MIAMI CHICAGO))

(DEFPARAMETER CITY-CONNECTIONS-LIST
  '((NEW-YORK (BOSTON 220) (WASHINGTON 500))
    (LOS-ANGELES (SAN-FRANCISCO 600) (WASHINGTON 2500))
    (BOSTON (NEW-YORK 220) (WASHINGTON 600) (CHICAGO 1000))
    (WASHINGTON (NEW-YORK 500) (BOSTON 600) (LOS-ANGELES 2500))
    (SAN-FRANCISCO (CHICAGO 1500) (LOS-ANGELES 600))
    (MIAMI (CHICAGO 2500))
    (CHICAGO (BOSTON 1000) (SAN-FRANCISCO 1500) (MIAMI 2500))))

```

```

;;;
;;; This is a hash table that contains the processor number assigned to each
;;; of the cities.
;;;
(DEFVAR PROCESSORS-FOR-CITIES (MAKE-HASH-TABLE))

;;;
;;; Here is the top level function
;;;
(DEFUN DO-FIND-SHORTEST-PATH (START-CITY STOP-CITY)
  (BUILD-GRAPH)
  (FIND-SHORTEST-PATH START-CITY)
  (PRINT-SHORTEST-PATH STOP-CITY))

(DEFVAR *NEXT-FREE-PROCESSOR* NIL "This is used to allocate processors")

;;;
;;; This function will load the graph into the CMS
;;;
(DEFUN BUILD-GRAPH ()
  ;; allocate more processors than we will need
  (*COLD-BOOT :INITIAL-DIMENSIONS '(100))

  ;; initialize all the processors in the machine to be neither connections
  ;; nor cities
  (*SET CITY-P NIL!!)
  (*SET CONNECTION-P NIL!!)

  ;; start allocating processors with processor 0
  (SETQ *NEXT-FREE-PROCESSOR* 0)

  ;; start by assigning processors to cities. All we have to do is write a T
  ;; into CITY-P, and write the name of the city into CITY-NAMES (this is
  ;; just a convenience)
  (LOOP FOR CITY IN LIST-OF-ALL-CITIES
    FOR CITY-PROCESSOR = *NEXT-FREE-PROCESSOR*
    DO
      (SETF (GETHASH CITY PROCESSORS-FOR-CITIES) CITY-PROCESSOR)
      (SETF (PREF CITY-P CITY-PROCESSOR) T)
      (SETF (PREF CITY-NAMES CITY-PROCESSOR) CITY)

      (INCF *NEXT-FREE-PROCESSOR*))

```

```

;; loop through all the cities, and set up their connections. This means
;; writing a T into CONNECTION-P, as well as the cities at both ends of the
;; connection, and the distance between them.
(LOOP FOR ITEM IN CITY-CONNECTIONS-LIST
  FOR CITY = (FIRST ITEM)
  FOR CONNECTIONS = (REST ITEM)
  DO
    (LOOP FOR CONNECTION IN CONNECTIONS
      FOR CONNECTED-CITY = (FIRST CONNECTION)
      FOR DISTANCE = (SECOND CONNECTION)
      FOR CONNECTION-PROCESSOR = *NEXT-FREE-PROCESSOR*
      DO
        (SETF (PREF CONNECTION-P CONNECTION-PROCESSOR) T)
        (SETF (PREF CONNECTION-DISTANCE CONNECTION-PROCESSOR) DISTANCE)

        (SETF (PREF CONNECTED-CITY-FROM CONNECTION-PROCESSOR)
          (GETHASH CITY PROCESSORS-FOR-CITIES))
        (SETF (PREF CONNECTED-CITY-TO CONNECTION-PROCESSOR)
          (GETHASH CONNECTED-CITY PROCESSORS-FOR-CITIES))

        (INCF *NEXT-FREE-PROCESSOR*)
      ))
  ;; End of building the graph routine
)

;;;
;;; Here is the actual algorithm for computing the length of the shortest path
;;; between two cities:
;;;
;;; We define one of the two cities as the START city and the other as the
;;; STOP city.
;;;
;;; (1) All cities set their distance from the START city to some very large
;;; number. The START city sets its distance to zero.
;;;
;;; (2) All connections fetch the distance pvar of their CONNECTED-CITY-FROM, and
;;; add on their CONNECTION-DISTANCE.
;;;
;;; (3) All connections send the result of the previous set to their
;;; CONNECTED-CITY-TO. The send is done with a :MIN combiner.
;;;
;;; (4) All cities set their distance from the START city to the minimum of their
;;; current distance and the value sent by the previous step.
;;;

```



```
;;; (5) If any city got a newer minimum distance, then go cycle back to step
;;; (2) again.
```

```
(DEFUN FIND-SHORTEST-PATH (START-CITY)
```

```
;; translate the start city into a processor number
  (SETQ START-CITY (GETHASH START-CITY PROCESSORS-FOR-CITIES))
```

```
;; (1) All cities set their distance from the START to some very large
;; number. The START city sets its distance to zero.
```

```
(*WHEN CITY-P (*SET CITY-DISTANCE-FROM-START (! 30000)))
  (SETF (PREF CITY-DISTANCE-FROM-START START-CITY) 0)
```

```
;; allocate some storage for the computation of the next CITY-DISTANCE-FROM-START
(*LET ((NEW-CITY-DISTANCE-FROM-START CITY-DISTANCE-FROM-START))
```

```
  (LOOP WITH ANY-NEW-DISTANCE-SHORTER-P ;This is T when we need to loop again.
    DO ;This will loop until the WHILE ...
      ;... below is false
```

```
;; (2) All connections fetch the distance pvar of their CONNECTED-CITY-FROM, and
;; add on their CONNECTION-DISTANCE.
```

```
(*WHEN CONNECTION-P
  (*LET ((DISTANCE-OF-CONNECTED-CITY
    (+!! CONNECTION-DISTANCE (PREF!! CITY-DISTANCE-FROM-START
      CONNECTED-CITY-FROM))))
```

```
;; (3) All connections send the result of the previous set to their
;; CONNECTED-CITY-TO. The send is done with a :MIN combiner.
```

```
(*PSET :MIN DISTANCE-OF-CONNECTED-CITY
  NEW-CITY-DISTANCE-FROM-START
  CONNECTED-CITY-TO)))
```

```
;;(4) All cities set their distance from the START city to the minimum of
;;    their current distance and the value sent by the previous step.
```

```
(*WHEN CITY-P
  (SETQ ANY-NEW-DISTANCE-SHORTER-P (*OR (<!! NEW-CITY-DISTANCE-FROM-START
    CITY-DISTANCE-FROM-START))))
  (*SET CITY-DISTANCE-FROM-START (MIN!! CITY-DISTANCE-FROM-START
    NEW-CITY-DISTANCE-FROM-START)))
```

```
;;(5) If any city got a newer minimum distance, then go cycle back to
;;    step (2) again.
```

```
WHILE ANY-NEW-DISTANCE-SHORTER-P))
```

```
;;;
;;; This function just simply pulls the distance of the STOP city from the
;;; CMS.
;;;
(DEFUN PRINT-SHORTEST-PATH (STOP-CITY)
  (FORMAT T "~% The distance from the START city to the stop city is ~d."
    (PREF CITY-DISTANCE-FROM-START
      (GETHASH STOP-CITY PROCESSORS-FOR-CITIES))))
```

Appendix B

List of ESSENTIAL *LISP Commands

This section contains a list of all ESSENTIAL *LISP commands described in this manual, grouped according to functionality. For a more detailed description of any command, refer to the indicated page number.

Configuration Constants:

number-of-processors-limit, page 11
log-number-of-processors-limit, page 11
number-of-dimensions, page 11
current-cm-configuration, page 11
t!!, page 11
nil!!, page 11

Pvar Operations:

*defvar symbol &optional pvar documentation-string, page 12
*deallocate-*defvars & rest pvar-names, page 13
allocate!! &optional pvar, page 13
*deallocate pvar, page 13
pvarp object, page 13

Temporary Allocation:

let ({(symbol &optional pvar)}) &rest body, page 14
 let ({(symbol &optional pvar)}*) &rest body, page 14

Setting the Value of Pvars:

set {pvar-1 pvar-2}, page 15

Reading and Writing the Memory:

pref pvar address, page 15
 (setf (pref pvar address) lisp-expression)
 pref-grid pvar &rest addresses, page 15
 (setf (pref-grid pvar &rest addresses) lisp-expression)

Selecting the Active Processors:

*all &rest body, page 16
 *when pvar &rest body, page 16
 *if pvar then-form &optional else-form, page 16
 cond {(pvar {form})}*, page 17
 with-css-saved {form}*, page 17
 do-for-selected-processors (symbol) &rest body, page 17

Predicate !! Functions:

=!! numeric-pvar &rest numeric-pvars, page 19
 /=! numeric-pvar &rest numeric-pvars, page 19
 <!! numeric-pvar &rest numeric-pvars, page 20
 >!! numeric-pvar &rest numeric-pvars, page 20

<=!! numeric-pvar &rest numeric-pvars, page 20

>=!! numeric-pvar &rest numeric-pvars, page 20

Bit Manipulation !! Functions:

lognot!! integer-pvar, page 20

logior!! &rest integer-pvars, page 20

logxor!! &rest integer-pvars, page 20

logand!! &rest integer-pvars, page 21

logeqv!! &rest integer-pvars, page 21

Boolean !! Functions:

not!! pvar, page 21

and!! &rest pvars, page 21

or!! &rest pvars, page 21

xor!! &rest pvars, page 21

eq1!! pvar1 pvar2, page 21

eq!! pvar1 pvar2, page 22

integerp!! pvar, page 22

floatp!! pvar, page 22

numberp!! pvar, page 22

zerop!! numeric-pvar, page 22

Numerical !! Functions

!! lisp-expression, page 22

+!! &rest numeric-pvars, page 22

-!! numeric-pvar &rest numeric-pvars, page 22

*!! &rest numeric-pvars, page 22

/!! numeric-pvar &rest numeric-pvars, page 23

1+!! numeric-pvar, page 23

1-!! numeric-pvar, page 23

min!! numeric-pvar &rest numeric-pvars, page 23

max!! numeric-pvar &rest numeric-pvars, page 23

mod!! numeric-pvar integer-pvar, page 23

ash!! integer-pvar count-pvar, page 23

truncate!! numeric-pvar &optional divisor-numeric-pvar, page 23

round!! numeric-pvar &optional divisor-numeric-pvar, page 23

ceiling!! numeric-pvar &optional divisor-numeric-pvar, page 23

floor!! numeric-pvar &optional divisor-numeric-pvar, page 24

sqrt!! non-negative-pvar, page 24

isqrt!! non-negative-integer-pvar, page 24

random!! limit-pvar, page 24

Miscellaneous !! Functions

load-byte!! from-pvar position-pvar size-pvar, page 24

deposit-byte!! into-pvar position-pvar size-pvar byte-pvar, page 24

if!! pvar then-pvar else-pvar, page 25

cond!! {(pvar {form}*)}*, page 25

enumerate!! pvar, page 25

rank!! numeric-pvar predicate, page 26

*User !! and * Functions:*

*defun, page 26

*funcall function &rest arguments, page 26

*apply function arg &optional more-args, page 26

Debugging Tools

pretty-print-pvar pvar &key mode format per-line start end, page 27

list-of-active-processors, page 27

pretty-print-pvar-in-currently-selected-set pvar &key format start end, page 27

Parallel Memory Operations:

pref!! pvar-expression cube-address-pvar, page 28
(setf (pref!! dest-pvar-expression cube-address-pvar) value-pvar)

pref-grid!! pvar-expression &rest grid-address-pvars &key border-pvar, page 28

pref-grid-relative!! pvar-expression &rest relative-address-pvars &key border-pvar,
page 29

*pset combiner value-pvar dest-pvar cube-address-pvar, page 29
 where combiner is one of :default, :overwrite, :or, :and, :logior, :logand, :add, :min, :max

*pset-grid combiner value-pvar dest-pvar &rest grid-address-pvars, page 30

*pset-grid-relative combiner value-pvar dest-pvar &rest relative-grid-address-pvars, page 30

Address Translation and Related Functions:

dimension-size dimension, page 11

self-address!!, page 32

self-address-grid!! dimension-pvar, page 32

grid-from-cube-address cube-address dimension, page 32

cube-from-grid-address address-pvar &rest address-pvars, page 32

grid-from-cube-address!! cube-address-pvar dimension-pvar, page 32

cube-from-grid-address!! address-pvar &rest address-pvars, page 33

off-grid-border-p!! &rest grid-address-pvars, page 33

off-grid-border-relative-p!! &rest relative-grid-address-pvars, page 33

Global Operations:

*logior integer-pvar, page 34

*logand integer-pvar, page 34

*min numeric-pvar, page 34

*max numeric-pvar, page 34

*or pvar, page 34

*and pvar, page 34

*sum numeric-pvar, page 34

Initialization:

*cold-boot &key initial-dimensions, page 35

*warm-boot, page 36

Functions that Exist Only in the Simulator:

display-*lisp-function-use-statistics, page 37

reset-*lisp-function-use-statistics, page 37

Bibliography

- [Hillis 85] W. Daniel Hillis. *The Connection Machine*. MIT Press (Cambridge, Massachusetts, 1985).
- [Lasser in preparation] Clifford A. Lasser *The Complete *Lisp Manual*. Thinking Machines Corporation (Cambridge, Massachusetts, 1986).
- [Steele 84] Guy L. Steele, Jr. *Common Lisp: The Language*. Digital Press (Billerica, Massachusetts, 1984).
- [Thinking Machines Corporation 86a] *Connection Machine Parallel Instruction Set (PARIS)*. Thinking Machines Corporation (Cambridge, Massachusetts, 1986).
- [Thinking Machines Corporation 86b] *Introduction to Data Level Parallelism*. Thinking Machines Corporation (Cambridge, Massachusetts, 1986).
- [Thinking Machines Corporation 86c] *Connection Machine User's Guide*. Thinking Machines Corporation (Cambridge, Massachusetts, 1986).

Index

- `*`, 6
- `/=!!`, 19
- `>=!!`, 20
- `-!!`, 22
- `/!!`, 23
- `!!` functions, 19
- `1+!!`, 23
- address, 5
 - cube, 5
 - grid, 5
- addressing, 32
 - processor, 32
- `*all`, 16
- `allocate!!`, 13
- allocating local pvars, 14
- `and!!`, 21
- `*and`, 34
- `*apply`, 26
- `ash!!`, 23

- border-pvar, 28-29

- `ceiling!!`, 23
- `*cold-boot`, 35, 39
- communication examples, 10
- component of a pvar, 6
- `*cond`, 17
- `cond!!`, 25
- configuration constants and functions, 10
- contents of a pvar, 5
- creating new pvars, 12
- cube address, 5
- cube-from-grid-address, 32
- cube-from-grid-address!!, 33
- `*current-cm-configuration*`, 11
- currently selected set, 6, 16

- `*deallocate`, 13

- *deallocate-*defvars, 13
- debugging tools, 27
- *defun, 26
- *defun examples, 9
- *defvar, 12
- deposit-byte!!, 24
- dimension-size, 11
- display-*lisp-function-use-statistics, 37
- do-for-selected-processors, 17

- enumerate!!, 25
- eq!!, 22
- eq!!, 21
- example programs, 49
- extent of pvars, 38

- field, 5
- floatp!!, 22
- floor!!, 24
- format of function definitions, 10
- *funcall, 26
- function definitions, 10
 - format of, 10
- functions, 19
 - !!, 19
 - logical !!, 20
 - miscellaneous !!, 24
 - numerical !!, 22
 - predicate !!, 19
 - user !! and *, 26

- global operations, 34
- grid address, 5
- grid-from-cube-address, 32

- hardware, 35
 - using, 35

- *if, 16
- if!!, 25
- initial-dimensions, 35
- integerp!!, 22
- isqrt!!, 24

- *let, 14
- *LISP package, 35
- list of **essential *lisp** commands, 50
- list-of-active-processors, 27

- load-byte!!, 24
- loap, 27
- logand!!, 21
- *logand, 34
- logeqv!!, 21
- logical !! functions, 20
- logical !! operations on numbers, 20
- logical !! operators, 21
- logior!!, 20
- *logior, 34
- lognot!!, 20
- *log-number-of-processors-limit*, 11
- logxor!!, 20

- mapcar, 39
 - use of on functions that return pvars, 39
- max!!, 23
- *max, 34
- min!!, 23
- *min, 34
- miscellaneous !! functions, 24
- mod!!, 23

- nil!!, 11
- not!!, 21
- *number-of-dimensions*, 11
- *number-of-processors-limit*, 11
- numberp!!, 22
- numerical !! functions, 22

- off-grid-border-p!!, 33
- off-grid-border-relative-p!!, 33
- operations, 34
 - global, 34
- operators, 21
 - logical !!, 21
- or!!, 21
- *or, 34
- overview of **essential *lisp**, 6

- parallel equivalent, 6
- parallel global memory references, 28
- potentially troublesome situations, 38
- ppp, 27
- ppp-css, 27
- pre-defined pvars, 11
- predicate !! functions, 19
- pref, 15

- pref!!, 28
- pref-grid, 15
- pref-grid!!, 28
- pref-grid-relative!!
- pretty-print-pvar, 27
- pretty-print-pvar-in-currently-selected-set, 27
- processor addressing, 28, 32
- processor selection, 16
- processors, 5
 - reading and writing fields in specific, 15
- *pset
- *pset-grid, 30
- *pset-grid-relative, 30
- pvar, 4-5, 12
 - component of a, 6
 - contents of a
- pvar data structure, 12
- pvar sample expressions, 6
- pvar values in non-selected processors, 38
- pvarp, 13
- pvars, 14
 - allocating local, 14
 - creating new, 12
 - extent of, 38
 - pre-defined, 11
 - setting the values of, 14
- random!!, 24
- rank!!, 26
- reading and writing fields in specific processors, 15
- reset-*lisp-function-use-statistics, 37
- round!!, 23
- selection examples, 8
- self-address!!, 32
- self-address-grid!!, 32
- *set, 15
- setf, 7, 15, 29-30
- setting the values of pvars, 14
- simulator, 37
 - using, 37
- size of the machine, 11
- sqrt!!, 24
- *sum, 34
- t!!, 11
- truncate!!, 23
- user !! and * functions, 26

using the hardware, 35
using the simulator, 37

virtual processors, 5

*warm-boot, 36, 39

*when, 16

with-css-saved, 17

xor!!, 21

zerop!!, 22

Thinking Machines Corporation

**Connection Machine
Parallel Instruction Set
(PARIS)**

The LISP Implementation

**Release 2, Revision 7
July 1986**

© 1986 Thinking Machines Corporation
All Rights Reserved

This notice is intended as a precaution against inadvertant publication and does not constitute an admission or acknowledgement that publication has occurred, nor constitute a waiver of confidentiality. The information and concepts described in this document are the proprietary and confidential property of Thinking Machines Corporation.

Document number 1-0002-2-7

© 1986 Thinking Machines Corporation.

“Connection Machine” is a registered trademark of Thinking Machines Corporation.

“*Lisp” and “PARIS” are trademarks of Thinking Machines Corporation.

“Symbolics 3600” and “Zetalisp” are trademarks of Symbolics, Inc.

“VAX” is a trademark of Digital Equipment Corporation.

This document corresponds to release 2, revision 7
of the Connection Machine Parallel Instruction Set (PARIS).

The information in this document is subject to change without notice and should not be construed as a commitment by Thinking Machines Corporation. Thinking Machines Corporation reserves the right to make changes to any products described herein to improve functioning or design. Although the information in this document has been reviewed and is believed to be reliable, Thinking Machines Corporation does not assume responsibility or liability for any errors that may appear in this document. Thinking Machines Corporation does not assume any liability arising from the application or use of any information or product described herein.

Contents

1	Introduction	1
2	Virtual Machine Architecture	2
2.1	Virtual Processor Organization	4
2.2	Memory	11
2.3	Stack	11
2.4	Flags	12
3	Data Formats	15
3.1	Bit Fields	15
3.2	Signed Integers	16
3.3	Unsigned Integers	16
3.4	Floating-Point Numbers	16
3.5	Processor Addresses	18
3.6	Configuration Variables	18
3.7	Implementation Restrictions	19
4	Operation Formats and Addressing Modes	22
4.1	Memory Addresses and the Stack	22
4.2	Unconditional Operations	23
4.3	Constant Operands	24
4.4	Flags as Operands	25
5	Getting Started	26
6	Simple Unary Operations	29
6.1	Unary Operations on Bit Fields	29
6.2	Unary Operations on Signed Integers	29
6.3	Unary Operations on Unsigned Integers	31
6.4	Unary Operations on Floating-Point Numbers	33
7	Simple Binary Operations	36
7.1	Binary Operations on Bit Fields	36
7.2	Binary Operations on Signed Integers	38

7.3	Binary Operations on Unsigned Integers	43
7.4	Binary Operations on Floating-Point Numbers	48
8	Other Simple Operations	50
8.1	Movement of Fields	50
8.2	Arrays	52
8.3	Miscellaneous Operations	53
9	Cooperative Computations	54
9.1	Global Operations	54
9.1.1	Global Operations on Bit Fields	54
9.1.2	Global Operations on Signed Integers	55
9.1.3	Global Operations on Unsigned Integers	55
9.1.4	Global Operations on Floating-Point Numbers	56
9.2	Enumeration	56
9.3	Sorting	58
10	Interprocessor Communication	59
10.1	General Communication through the Router	59
10.2	Communication through the NEWS Grid	63
10.3	Addresses and Address Transformations	64
11	Memory Data Transfers	66
11.1	Transfer of Single Items	66
11.2	Transfer of Arrays	67
12	Housekeeping Operations	70
12.1	Stack Limit, Pointer, and Upper Bound	70
12.2	Initializing the Random Number Generator	71
12.3	Controlling the Cabinet Lights (LEDs)	71
12.4	Initializing the Connection Machine System	71
12.5	Getting Information	78

List of Figures

2.1	65,536 processors	3
2.2	The NEWS coordinates for a 128×512 grid of processors	5
2.3	Cube addresses for a 128×512 virtual grid on 65,536 physical processors	7
2.4	Cube addresses for a 128×512 virtual grid on 32,768 physical processors	8
2.5	Cube addresses for a 128×512 virtual grid on 16,384 physical processors	9
2.6	Cube addresses for a 128×512 virtual grid on 8,192 physical processors	10
2.7	Virtual Processor Memory Layout and Flags	11
12.1	Microcontrollers, Front Ends, and Nexus	73

Chapter 1

Introduction

PARIS is a low-level instruction set for programming the Connection Machine computer system. It is the lowest-level protocol by which the actions of Connection Machine processors are directed by the front-end computer. PARIS is sometimes referred to as a “macroinstruction set” for the Connection Machine system because it is comparable in power to the (macro)instruction sets of typical sequential processors such as the VAX, and to distinguish it from the “microinstruction set” (microcode) that is executed by the Connection Machine system microcontroller and the “nanoinstruction set” that is directly executed by the individual hardware Connection Machine processors.

PARIS is intended primarily as a base upon which to build higher-level languages for the Connection Machine system. It provides a large number of operations similar to the machine-level instruction set of an ordinary computer. PARIS supports primitive operations on signed and unsigned integers and floating-point numbers, as well as message-passing operations and facilities for transferring data between the Connection Machine processors and the front-end computer.

The PARIS user interface consists of a set of macros, functions, and variables to be called from user code. The macros and functions direct the actions of the Connection Machine system by sending macroinstructions to the Connection Machine microcontroller, and the variables allow the user program to find out information about the Connection Machine system such as the number of processors available.

Several different versions of the user interface are provided, one for the Lisp programming language and others for other languages. These interfaces are functionally identical; they differ only in conforming to the syntax and data types of the one language or the other. The description of PARIS is presented using the Lisp syntax and assuming use of the Lisp language. Definitions of PARIS operations are presented in the style of the book *Common Lisp: The Language* by Steele et al. (Digital Press, 1984).

Many PARIS operations are implemented directly by the microcontroller in microcode; such operations are mapped one-to-one onto microcontroller directives. Other PARIS operations are more complicated, and require some cooperation between the microcontroller and library routines that run in the front-end computer.

Chapter 2

Virtual Machine Architecture

An important property of the Connection Machine architecture is *scalability*. At present a single Connection Machine system can have 16,384 or 32,768 or 65,536 physical (hardware) processors, of which any single user can use a portion containing 8,192 or 16,384 or 32,768 or 65,536 processors. (See figure 2.1 for an illustration of 65,536 processors.) In most cases the same software can be executed unchanged on Connection Machine systems (or portions) with different numbers of physical processors; the number of processors affects only the size of the problem that can be handled.

PARIS enhances this scalability by presenting to the user an abstract version of the Connection Machine hardware. The most important feature is the *virtual processor* facility, whereby each physical processor is used to simulate some number of virtual processors. The point is that a program can be written assuming any appropriate number of processors; these virtual processors are then mapped onto physical processors. In this way a program can be executed unchanged on Connection Machine systems with different numbers of physical processors, even if it requires a certain minimum number of processors, with an essentially linear trade-off between number of physical processors and execution time. (There is a memory trade-off as well: the memory of a physical processor is divided equally among the virtual processors it supports.)

The Connection Machine hardware supports two mechanisms for interprocessor communication. The more general mechanism is the *router*, which allows data to be sent from any processor directly to any other processor; indeed, many processors can send data to many other processors simultaneously. The less general mechanism is redundant, but optimizes an important special case for speed. It organizes the processors as a two-dimensional grid and allows every processor to send data to an immediate neighbor in one of four directions, labelled North, East, West, and South; this mechanism is called the NEWS grid, from the initials of the four directions. Using these hardware mechanisms, PARIS provides identical virtual mechanisms within the virtual-processor framework.

The Connection Machine hardware provides a very primitive instruction set, where each instruction operates on only a few bits per processor. PARIS implements a rich virtual instruction set, including arithmetic on integer and floating-point number representations. PARIS also supports a per-processor stack, indexed by a global (not per-

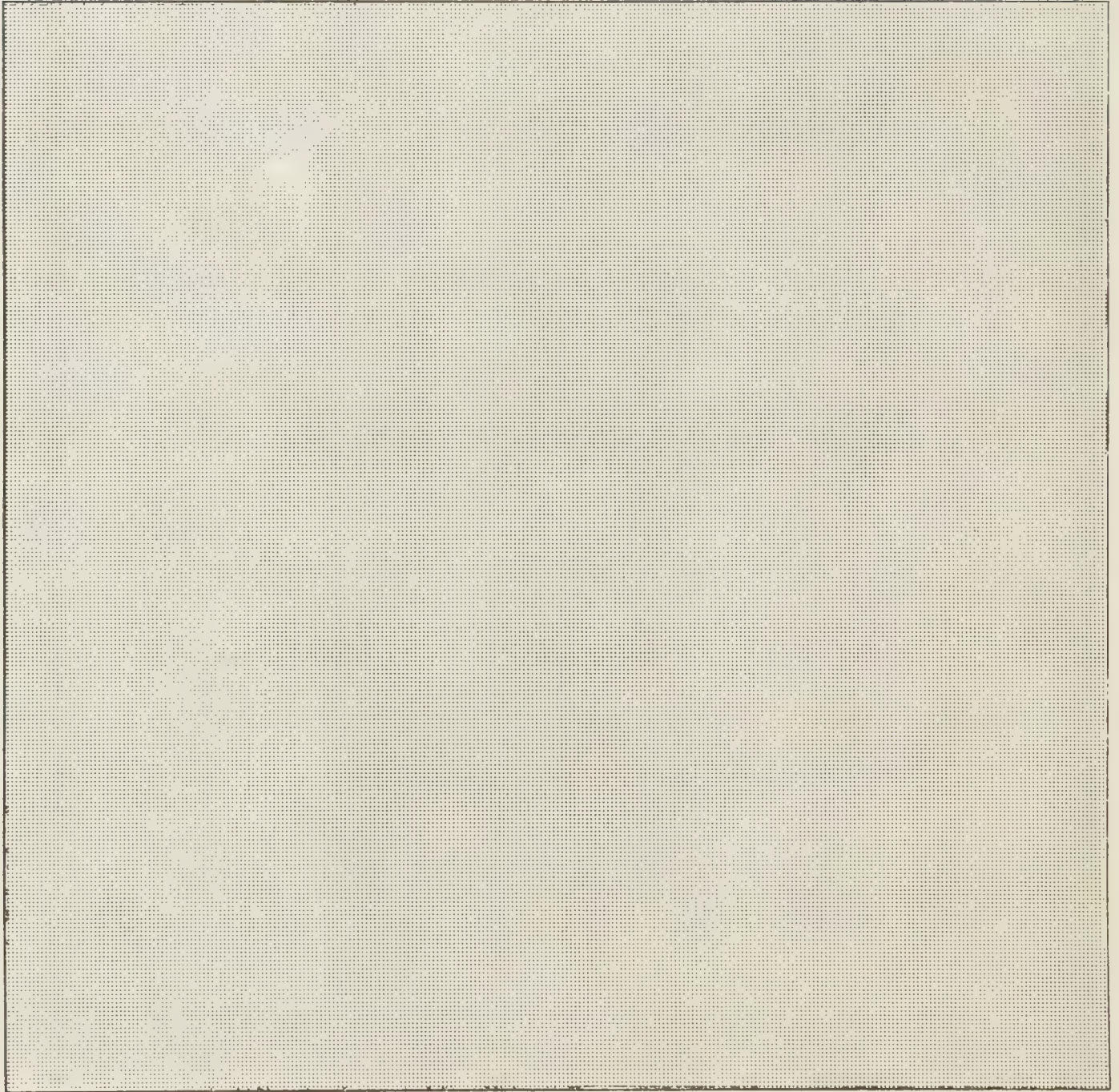


Figure 2.1: 65,536 processors

processor) stack pointer. The PARIS instruction set is comparable in expressive power to the assembly language of an ordinary computer, except that a single PARIS instruction can call for parallel execution of many copies of an operation, one per processor.

2.1 Virtual Processor Organization

When a PARIS program is to be run on the Connection Machine system, certain initialization operations must be performed first (see `cm:cold-boot`). Among other things, the virtual processor configuration must be defined. Two numbers specify the number of virtual processors per physical processor. Each physical processor simulates a small two-dimensional grid of virtual processors; the two numbers indicate the size of this grid in the *x* (east-west) and *y* (north-south) directions. These small grids are then implicitly joined together to make one large NEWS grid, in exactly the same way that the physical processors are arranged as a two-dimensional grid. The variables `cm:*x-virtual-to-physical-processor-ratio*` and `cm:*y-virtual-to-physical-processor-ratio*` are set by the initialization program to the size of the small grid in the *x* and *y* directions.

For example, suppose that the numbers 2 and 4 are specified. Then each physical processor will simulate 8 virtual processors arranged in a 2×4 grid. The 65,536 physical processors of an entire Connection Machine system are arranged in a 128×512 grid, so the entire virtual grid will be of size

$$(2 \times 128) \times (4 \times 512) = 256 \times 2048$$

for a total of 524,288 (512K) virtual processors.

One might instead specify the numbers 8 and 2. Then each physical processor will simulate 16 virtual processors. The entire virtual grid will be of size

$$(8 \times 128) \times (2 \times 512) = 1024 \times 1024$$

for a total of 1,048,576 (1M) virtual processors.

If the numbers 1 and 1 are specified, then there is one virtual processor per physical processor; as far as execution speed is concerned, it is as if virtual processors were not in use.

Each virtual processor can be identified by its NEWS coordinates (*x*, *y*). The *x* coordinate increases toward the east, and the *y* coordinate increases toward the south, *not* toward the north. This convention is chosen to be compatible with the coordinate system used for raster-scan display terminals, where north is up, south is down, east is right, and west is left, *x* increases rightward, and *y* increases downward. See figure 2.2.

Virtual processors can communicate not only via the virtual NEWS grid, but also via message-sending through the Connection Machine routers. Every virtual processor (like every physical processor) is labelled by a distinct integer called its *cube address*, or sometimes just *address*. Virtual processor addresses are in general longer than physical processor addresses, but otherwise behave in much the same way. The global variable

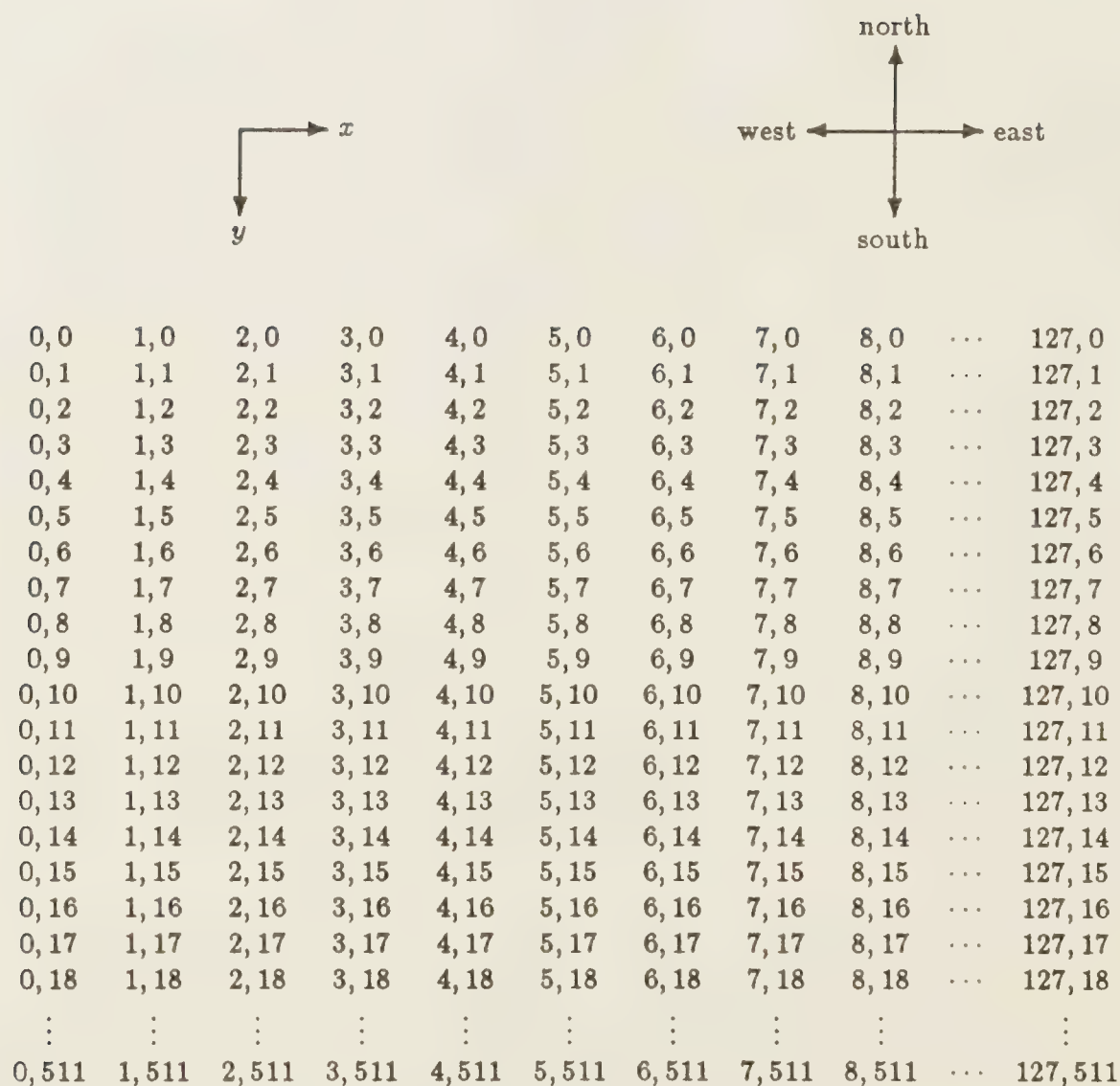
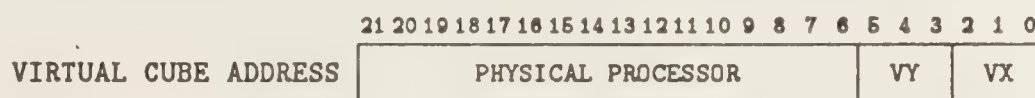


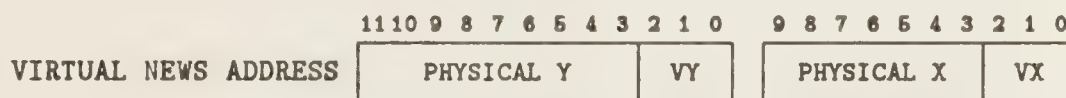
Figure 2.2: The NEWS coordinates for a 128×512 grid of processors. The x coordinate increases toward the east, and the y coordinate increases toward the south. This is the standard NEWS configuration when there are 65,536 physical processors and one virtual processor per physical processor.

`cm:cube-address-length` specifies how many bits are needed to represent a virtual processor address. This number is equal to the number of bits per physical processor address plus $\log_2 x + \log_2 y$, where x and y are the values of `cm:x-virtual-to-physical-processor-ratio` and `cm:y-virtual-to-physical-processor-ratio`. These values must always be integral powers of two.

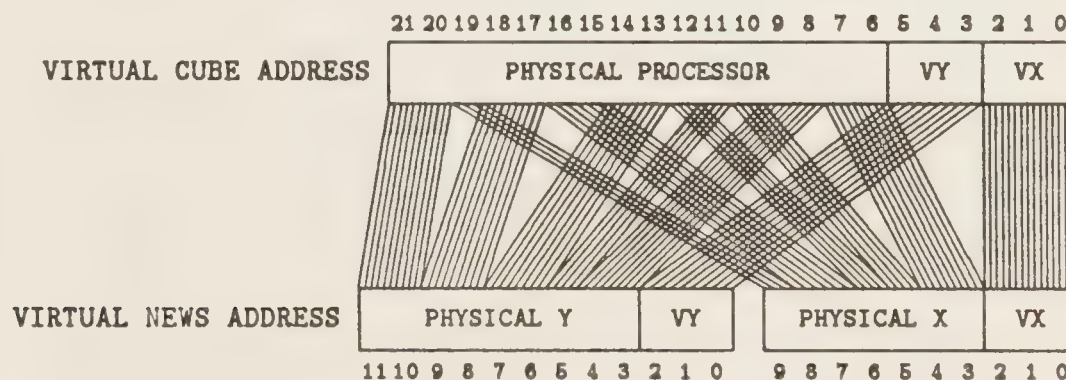
To be more specific, let us assume that a Connection Machine system with 65,536 physical processors is in use and that the virtual grid size is 8×8 . Then 3 bits are needed to represent a virtual x-address VX and 3 bits to represent a virtual y address VY within each processor. A virtual cube address is then laid out in this manner:



Furthermore virtual x and y addresses are laid out in this manner (the y address is shown on the left, and the x address on the right):



The mapping between the two looks like this:



The permutation of virtual NEWS address bits that produces the virtual cube address is dependent on the number of physical processors being used to support the given number of virtual processors. (See figures 2.3, 2.4, 2.5, and 2.6.) It is also dependent on the specific hardware implementation of the physical NEWS grid. It is therefore best to avoid dependence on the particular permutation; always use the operations `cm:x-from-cube`, `cm:y-from-cube`, and `cm:cube-from-x-y` to translate between cube addresses and NEWS addresses.

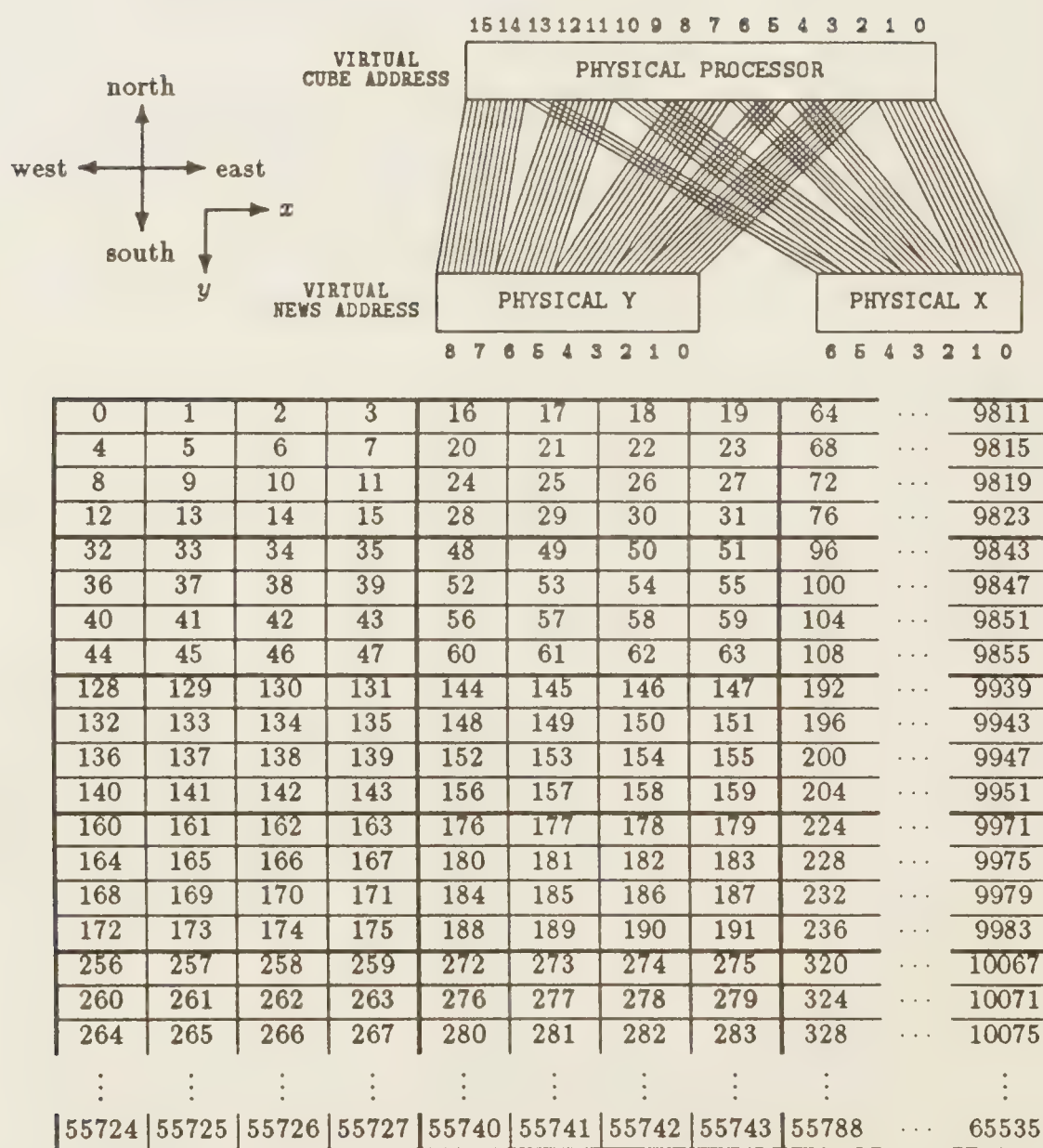


Figure 2.3: The cube addresses for a 128×512 grid of processors using 65,536 physical processors. Comparing this with figure 2.2 reveals the mapping between cube addresses and NEWS coordinates in the case where 65,536 physical processors are used to contain a 128×512 NEWS grid. There is one virtual processor per physical processor. Each box represents one physical processor; thick lines delimit groups of 16 physical processors.

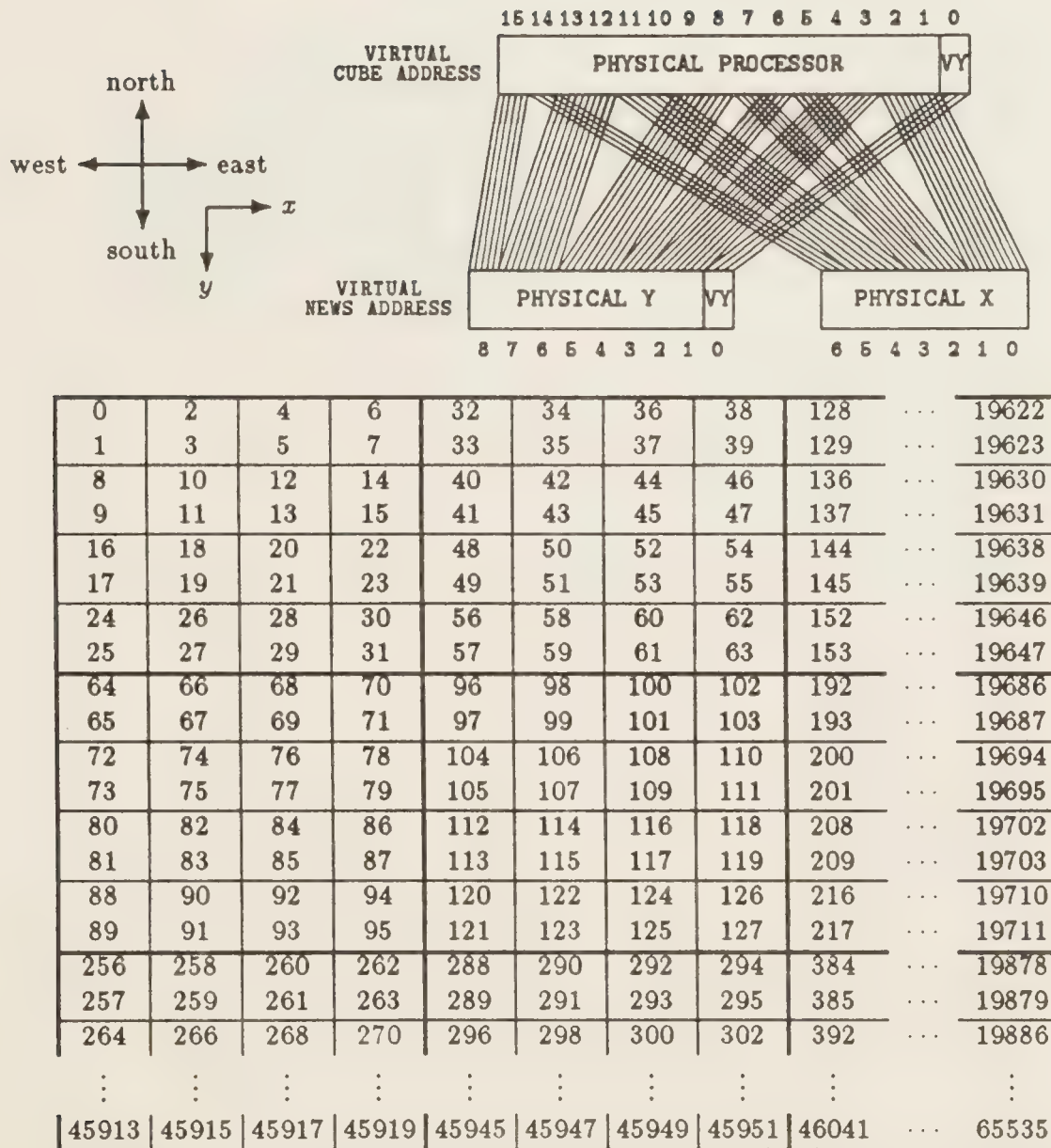


Figure 2.4: The cube addresses for a 128×512 grid of processors using 32,768 physical processors. In this case there is a 1×2 grid of virtual processors per physical processor. Each box represents one physical processor; thick lines delimit groups of 16 physical processors.

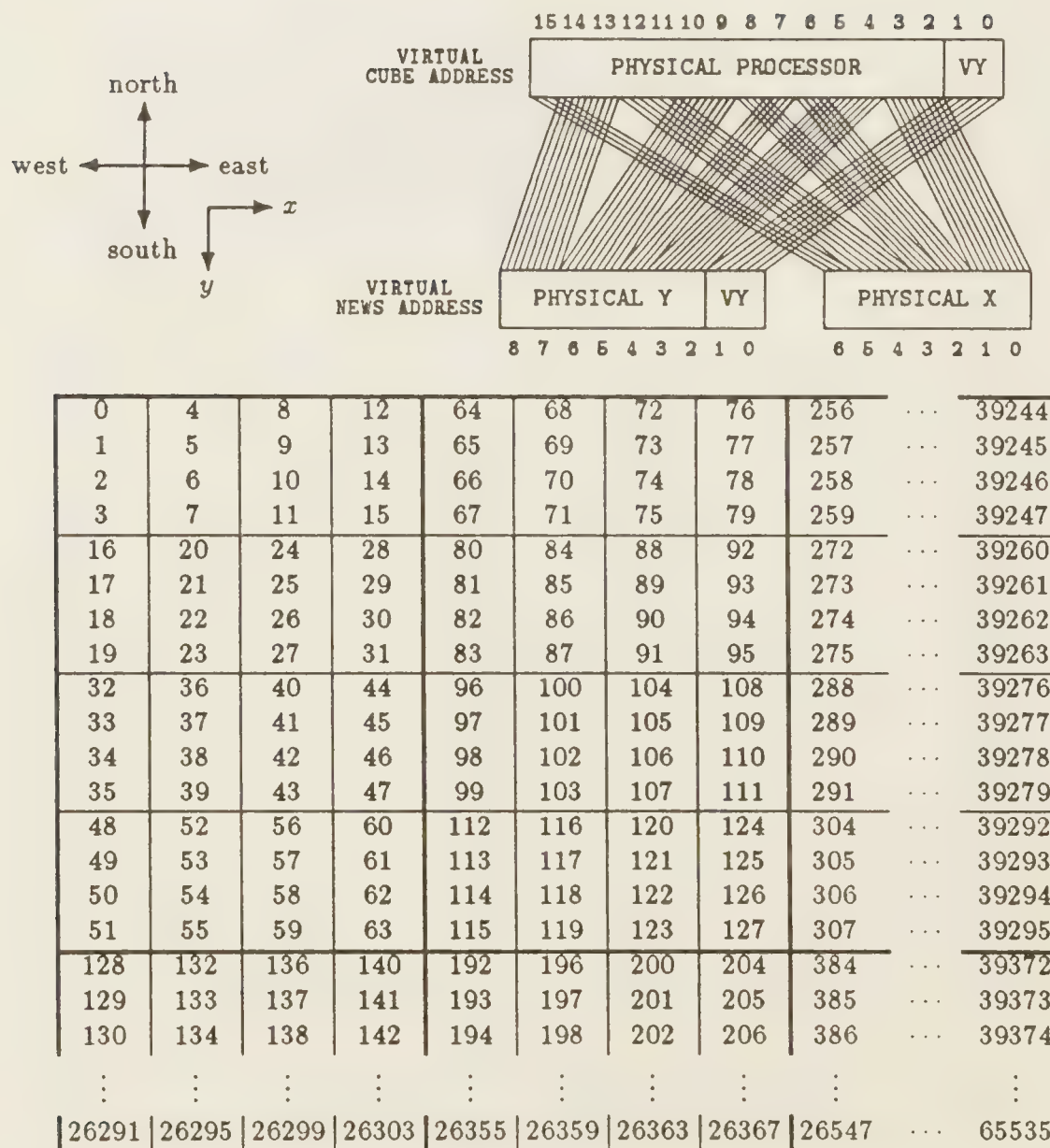


Figure 2.5: The cube addresses for a 128×512 grid of processors using 16,384 physical processors. In this case there is a 1×4 grid of virtual processors per physical processor. Each box represents one physical processor; thick lines delimit groups of 16 physical processors.

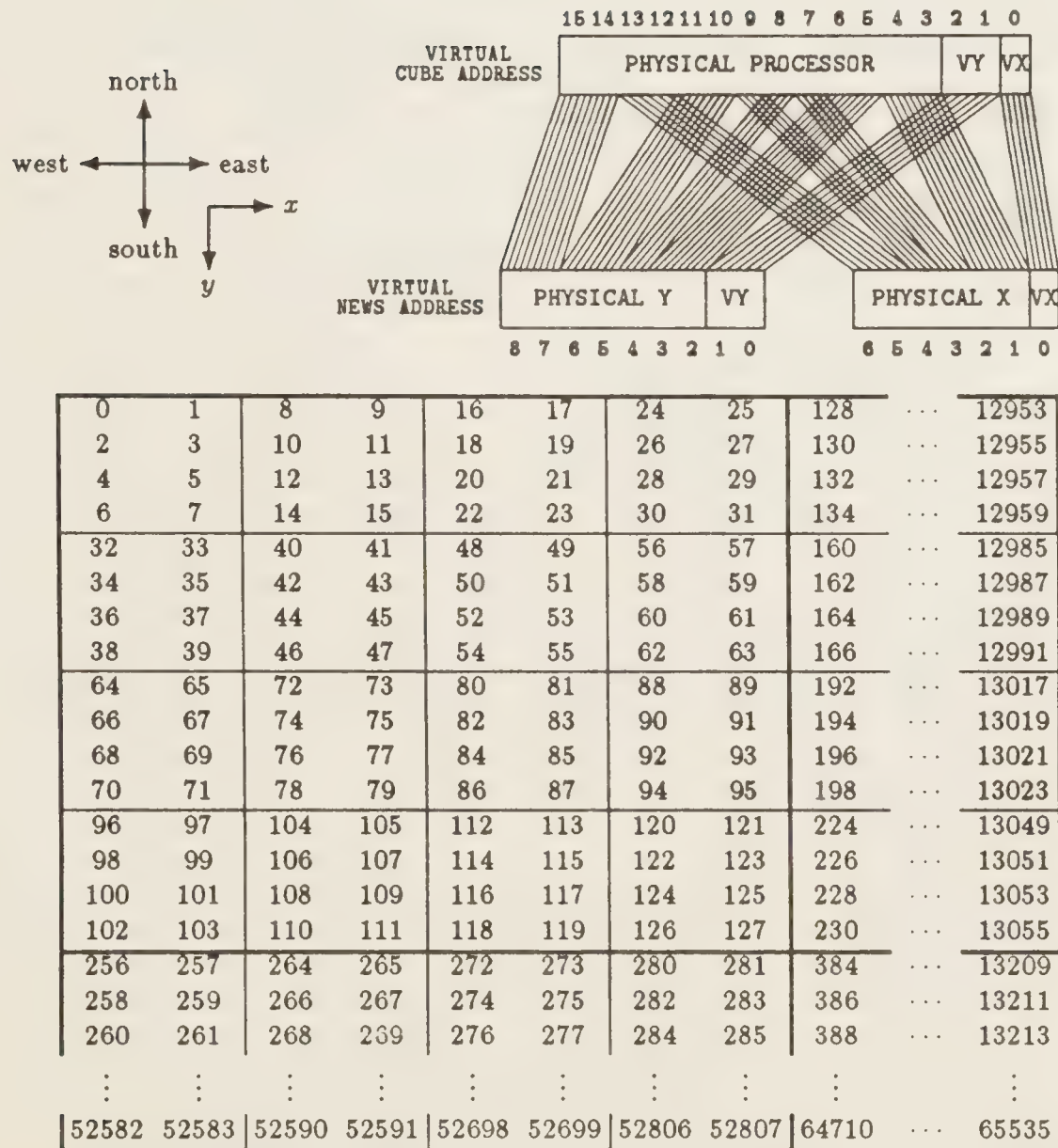


Figure 2.6: The cube addresses for a 128×512 grid of processors using 8,192 physical processors. In this case there is a 2×4 grid of virtual processors per physical processor. Each box represents one physical processor; thick lines delimit groups of 16 physical processors.

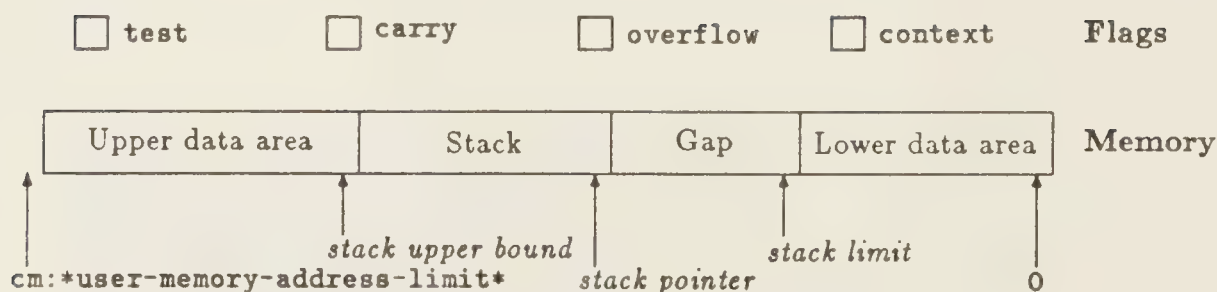


Figure 2.7: Virtual Processor Memory Layout and Flags

2.2 Memory

Each virtual processor has some amount of memory, addressible down to individual bits. Within each virtual processor the bit addresses run from 0 (inclusive) to some limit m (exclusive); the limit m is the value of the global variable `cm:*user-memory-address-limit*`. Typically the memory size m ranges from a few dozen to a few thousand bits, depending on how many virtual processors there are per physical processor.

In principle any part of this memory may be used for any purpose. However, PARIS conventionally divides the memory of each virtual processor into three regions: the *data area*, the *gap*, and the *stack*. The data area in turn may be divided into two parts, one below the stack and gap, and one above them. See figure 2.7. The areas are delimited by the *stack limit*, the *stack pointer*, and the *stack upper bound*. The lower data area runs from address 0 (inclusive) to the stack limit (exclusive), and may be used for arbitrary purposes by the user. The gap runs from the stack limit (inclusive) to the stack pointer (exclusive), and is an expansion area; the stack implicitly grows into it as necessary. The data area may also be expanded into the gap, or reduced, returning space to the gap, by explicitly resetting the stack limit pointer using the `cm:set-stack-limit` operation. The stack runs from the stack pointer (inclusive) to the stack upper bound (exclusive); the upper bound is a boundary beyond which the stack should not be popped. The upper data area runs from the stack upper bound (inclusive) to the end of memory for that virtual processor. In many applications the upper data area is empty, but the existence of a stack upper bound distinct from the end of memory allows some flexibility in placing or moving around the stack area. In most applications the stack limit and stack upper bound are initialized once, or change only infrequently, and the stack pointer varies between them as items are pushed and popped. The stack pointer points to the “top” of the stack (which is at the lower-addressed end of the stack area).

2.3 Stack

The stack begins near the high end of memory within each virtual processor and grows downward. Most PARIS operations take operands from absolute memory locations. Stack-relative addressing may also be used, wherein an operand is addressed relative to

the stack pointer (but no implicit pushing or popping is performed). There is a single global stack pointer, rather than one per processor. A number of operations are provided for moving the stack pointer and changing the stack limit and stack upper bound; see section 12.1.

A few PARIS operations require the implicit use of the stack (actually, the gap) for temporary scratch memory:

<code>cm:aref</code>	<code>cm:aset</code>
<code>cm:global-add</code>	<code>cm:max-scan</code>
<code>cm:plus-scan</code>	<code>cm:unsigned-max-scan</code>
<code>cm:processor-cons</code>	<code>cm:float-max-scan</code>
<code>cm:store-with-overwrite</code>	<code>cm:get-from-north</code>
<code>cm:store-with-logior</code>	<code>cm:get-from-east</code>
<code>cm:store-with-logand</code>	<code>cm:get-from-west</code>
<code>cm:store-with-logxor</code>	<code>cm:get-from-south</code>
<code>cm:store-with-add</code>	<code>cm:rank</code>
<code>cm:store-with-max</code>	<code>cm:unsigned-rank</code>
<code>cm:store-with-min</code>	<code>cm:float-rank</code>
<code>cm:store-with-unsigned-max</code>	<code>cm:store</code>
<code>cm:store-with-unsigned-min</code>	<code>cm:fetch</code>
<code>cm:get</code>	

Such operations will signal an error if the gap is not large enough to provide the required amount of temporary stack storage. See the individual descriptions of these operations for formulas describing the amount of storage required. Note that these operations may alter memory in the gap of any processor, whether selected or not.

2.4 Flags

Each PARIS virtual processor has an assortment of one-bit flags. Many PARIS operations store into these flags rather than, or in addition to, storing results into the memory. For example, the `cm:+` operation adds one signed integer to another, but also stores information into the `carry` flag and `overflow` flag.

The entire set of flags for each virtual processor is as follows:

- The `context` flag indicates which processors are active. Nearly all PARIS operations are *conditional*; the operation is effectively carried out only in those processors whose `context` flag is 1, and processors whose `context` flag is 0 are unaffected. Some operations are always unconditional. *All PARIS operations described below are conditional unless the individual description states otherwise.*
- The `overflow` flag indicates which operations produced results that the destination field was too small to contain. Many PARIS operations can affect the `overflow` flag. As a rule, integer operations set or clear the `overflow` flag, while floating-point operations are “sticky” in that they either set the `overflow` flag or else leave it unchanged.

- The **carry flag** holds the carry in and carry out for some arithmetic operations. The following operations can affect the carry flag:

cm:+	cm:u+	cm:-	cm:u-
cm:+constant	cm:u+constant	cm:-constant	cm:u-constant
cm:+carry	cm:u+carry	cm:-borrow	cm:u-borrow
cm:+flags	cm:u+flags		

Only the operations `cm:+carry`, `cm:u+carry`, `cm:-borrow`, and `cm:u-borrow` use the carry flag as an implicit input.

- The **test flag** holds the result of numeric comparisons and other tests, or indicates which operations failed because of bad operands. The following operations can affect the test flag:

cm:isqrt		cm:float-sqrt
cm:zerop	cm:unsigned-zerop	cm:float-zerop
cm:minusp		cm:float-minusp
cm:plusp	cm:unsigned-plusp	cm:float-plusp
		cm:f/
cm:max	cm:unsigned-max	cm:float-max
cm:min	cm:unsigned-min	cm:float-min
cm:max-constant	cm:unsigned-max-constant	
cm:min-constant	cm:unsigned-min-constant	
cm:=	cm:u=	cm:f=
cm:/=	cm:u/=	cm:f/=
cm:<	cm:u<	cm:f<
cm:<=	cm:u<=	cm:f<=
cm:>	cm:u>	cm:f>
cm:<=	cm:u<=	cm:f<=
cm:=constant	cm:u=constant	
cm:/=constant	cm:u/=constant	
cm:<constant	cm:u<constant	
cm:<=constant	cm:u<=constant	
cm:>constant	cm:u>constant	
cm:<=constant	cm:u<=constant	
cm:floor-divide	cm:unsigned-floor-divide	
cm:ceiling-divide	cm:unsigned-ceiling-divide	
cm:truncate-divide	cm:unsigned-truncate-divide	
cm:round-divide	cm:unsigned-round-divide	
cm:mod	cm:unsigned-mod	
cm:rem	cm:unsigned-rem	
cm:floor-and-mod	cm:unsigned-floor-and-mod	
cm:ceiling-and-remainder	cm:unsigned-ceiling-and-remainder	
cm:truncate-and-rem	cm:unsigned-truncate-and-rem	

cm:round-and-remainder	cm:unsigned-round-and-remainder
cm:global-max	cm:unsigned-global-max
cm:global-min	cm:unsigned-global-min
cm:aref	cm:aset
cm:send-with-overwrite	cm:store-with-overwrite
cm:send-with-logior	cm:store-with-logior
cm:send-with-logand	cm:store-with-logand
cm:send-with-logxor	cm:store-with-logxor
cm:send-with-add	cm:store-with-add
cm:send-with-max	cm:store-with-max
cm:send-with-min	cm:store-with-min
cm:send-with-unsigned-max	cm:store-with-unsigned-max
cm:send-with-unsigned-min	cm:store-with-unsigned-min
cm:send	cm:store

Chapter 3

Data Formats

The memory of a Connection Machine processor may be regarded as a simple linear sequence of bits, with no particular alignment or grouping characteristics. A data item always consists of a string of bits having consecutive addresses within the memory of a processor. Such a bit string is called a *field*.

Many PARIS operations may be regarded as interpreting bit fields as being of particular data types or formats. Currently PARIS supports three data types besides ordinary bit fields:

- signed integers, represented in two's-complement format
- unsigned integers, represented in straight binary format
- floating-point numbers, represented in a format close to that specified by IEEE standard 754 for floating-point arithmetic

The Connection Machine system allows unusual flexibility in that the hardware does not enforce any particular length or alignment requirements. PARIS supports integers and floating-point numbers of almost any size.

Most PARIS operations operate on fields within a processor, delivering results to other fields within that processor. Frequently we speak of one data item, but really mean to speak of many instances of that data item, one for each selected processor, to be considered or operated on in parallel. For example, when we say that an operation sets a flag when a field has such-and-so value, we mean that a separate decision is made within each processor whether to set that processor's flag, based on the value of the field within that processor.

3.1 Bit Fields

A bit field is specified by a bit address a and a positive length n ; the field consists of the bits with addresses a through $a + n - 1$, inclusive. Therefore the address of a field is the same as that of the lowest-addressed bit.

Some PARIS operations can operate on the flags as if they were bit fields of length 1 residing in memory. The permitted operations on flags are described in section 4.4.

3.2 Signed Integers

A signed integer is specified in the same way as a simple bit field, by a bit address a and a positive length n . The signed integer is represented in two's-complement form, and so a signed integer of length n can take on values in the range $-(2^{n-1})$ through $2^{n-1} - 1$, inclusive. The least significant bit has address a , and the most significant (sign) bit has address $a + n - 1$.

All arithmetic on signed integers is performed in a strict wraparound mode. As a rule, if the result of an operation overflows the destination field, the **overflow** flag is set, and the destination receives as many low-order bits of the true result as will fit. For example, using 4-bit signed arithmetic, multiplying 4 by -7 will produce the 4-bit result 4 (and also set the **overflow** flag), because the two's-complement representation of -28 is $\dots 11111100100$, of which the four low-order bits are 0100, or 4. Signed-integer operations that do not overflow clear the **overflow** flag.

In order to simplify the Connection Machine microcode, this arbitrary restriction is imposed: the length n may not be zero or one. In addition, certain operations on signed integers cannot handle operands whose length is greater than the value of the variable `cm:*maximum-integer-length*`; see section 3.7.

3.3 Unsigned Integers

An unsigned integer is specified in the same way as a simple bit field, by a bit address a and a positive length n . The unsigned integer is represented in straight binary form, and so an unsigned integer of length n can take on values in the range 0 through $2^n - 1$, inclusive. The least significant bit has address a , and the most significant bit has address $a + n - 1$.

All arithmetic on unsigned integers is performed in a strict wraparound mode, modulo 2^n . As a rule, if the result of an operation overflows the destination field, the **overflow** flag is set, and the destination receives as many low-order bits of the true result as will fit. For example, using 4-bit unsigned arithmetic, multiplying 4 by 7 will produce the 4-bit result 12 (and also set the **overflow** flag), because the two's-complement representation of 28 is $\dots 00000011100$, of which the four low-order bits are 1100, or 12. Unsigned-integer operations that do not overflow clear the **overflow** flag.

Unsigned integers, unlike signed integers, may be of length zero or one as well as of larger sizes. However, certain operations on unsigned integers cannot handle operands whose length is greater than the value of the variable `cm:*maximum-integer-length*`; see section 3.7.

3.4 Floating-Point Numbers

A floating-point data item is specified by three parameters: a bit address a , a significand length s , and an exponent length e . The total number of bits in the representation is $s + e + 1$, and the data item occupies the bits with addresses a through $a + s + e$, inclusive.

The significand occupies bits a through $a+s-1$, with the least significant bit at address a . A hidden-bit representation is used, and so the significand is normally interpreted as having a 1-bit as its most significant bit implicitly just above the bit at address $a+s-1$. If the exponent field is all zero-bits, however, then the hidden bit is taken to be 0.

The exponent occupies bits $a+s$ through $a+s+e-1$, with the least significant bit at address $a+s$. An excess- $(2^{e-1}-1)$ representation is used.

The sign bit occupies bit $a+s+e$, and is 1 for a negative number and 0 for a positive number. Overall, a sign-magnitude representation is used, so inverting the sign of a floating-point number merely involves flipping the sign bit. Note that there is both a plus zero and a minus zero.

When $s = 23$ and $e = 8$, this is equivalent to the IEEE standard single-precision format, which looks like this:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	exponent										significand																				

All of the PARIS floating-point operations default their floating-point length parameters to this format. When $s = 52$ and $e = 11$, the PARIS floating-point format is equivalent to IEEE standard double-precision format. The IEEE standard single-extended and double-extended formats can also be accommodated by suitable choices of s and e .

While the PARIS floating-point *format* is equivalent to the IEEE standard format, it must be emphasized that the PARIS implementation does not support equivalent *operations* at this time.¹ “Soft” underflow (using denormalized numbers for the result) is not supported. Rounding is performed correctly in all cases, using the round-to-nearest mode; the several rounding modes are not supported. The not-a-number (NaN) values are not supported. The standard exceptions and flags are not all supported. It is strongly recommended that a user of PARIS always use the IEEE standard formats unless careful analysis of the application (such as a need for speed or additional exponent range) indicates that another format is required and adequate.

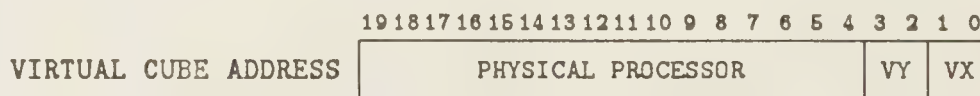
As a rule, if the result of an operation overflows, the overflow flag is set, and the destination exponent receives as many low-order bits of the true result exponent as will fit. Operations that do not overflow leave the overflow flag unaffected; they do not clear it.

In order to simplify the Connection Machine microcode, these arbitrary restrictions are imposed: $e \geq 2$, $s \geq 1$, and $2^{e-1} \geq s + 1$. In addition, certain operations on floating-point numbers cannot handle operands for which s exceeds the value of the variable `cm:*maximum-significand-length*` or e exceeds the value of the variable `cm:*maximum-exponent-length*`; see section 3.7.

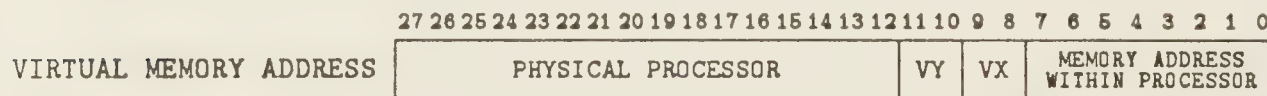
¹Thinking Machines Corporation does intend to support all standard IEEE arithmetic operations in a future software release.

3.5 Processor Addresses

Two kinds of addresses are used in PARIS for general interprocessor communication. The first is the *virtual cube address*; such an address identifies an individual virtual processor. The number of bits in a virtual cube address is specified by the variable `cm:cube-address-length*`. Such an address has this form for a 4×4 virtual NEWS configuration:



The second kind of address is the *virtual memory address*, which is the concatenation of a virtual cube address (in the more significant bits) and the address of a bit within the memory of the specified processor (in the less significant bits). A virtual memory address therefore points to a specific bit or field within the entire Connection Machine system. Such a virtual memory address has this form for a 4×4 virtual NEWS configuration:



Such addresses usually reside in the memory of a Connection Machine processor as bit fields. To specify such an address to a PARIS operation, one gives the address (as a source or destination) of the bit field that contains the address.

3.6 Configuration Variables

The current configuration of the machine is reflected in global variables. Programs refer to these so they can adapt to various sizes of machine. These variables are set by `cm:cold-boot`. They should never be set by the user, as there are many dependencies between them, which, if violated, will result in errors. Some variables are fixed by the hardware, while others depend on the arrangement of virtual processors set up by `cm:cold-boot`.

Let p and q be the x and y dimensions, respectively, of the physical NEWS grid connecting all the physical processors. Let v and w be the x and y dimensions, respectively, of the overall virtual NEWS grid connecting all the virtual processors. Each of these four quantities is an integral power of two, and furthermore $v \geq p$ and $w \geq q$.

Let m be the amount of the memory in each virtual processor available to the user. (This amount will not necessarily be a power of two.) Every memory address a within a virtual processor must be in the range $0 \leq a < m$.

The values of most of the configuration variables are expressible in terms of p , q , v , w , and m as shown below.

cm:*user-memory-address-limit* = m	[Variable]
cm:*user-cube-address-limit* = $v \cdot w$	[Variable]
cm:*full-virtual-memory-address-limit* = $m \cdot v \cdot w$	[Variable]
cm:*user-x-dimension-limit* = v	[Variable]
cm:*user-y-dimension-limit* = w	[Variable]
cm:*physical-cube-address-limit* = $p \cdot q$	[Variable]
cm:*physical-x-dimension-limit* = p	[Variable]
cm:*physical-y-dimension-limit* = q	[Variable]
cm:*virtual-to-physical-processor-ratio* = $(v \cdot w)/(p \cdot q)$	[Variable]
cm:*x-virtual-to-physical-processor-ratio* = v/p	[Variable]
cm:*y-virtual-to-physical-processor-ratio* = w/q	[Variable]

These quantities are useful for measuring numbers of processors and memory locations.

cm:*user-memory-address-length* = $\lceil \log_2 m \rceil$	[Variable]
cm:*cube-address-length* = $\log_2(v \cdot w)$	[Variable]
cm:*virtual-memory-address-length* = $\lceil \log_2 m \cdot v \cdot w \rceil$	[Variable]
cm:*x-news-address-length* = $\log_2 v$	[Variable]
cm:*y-news-address-length* = $\log_2 w$	[Variable]
cm:*physical-cube-address-length* = $\log_2(p \cdot q)$	[Variable]
cm:*physical-x-news-address-length* = $\log_2 p$	[Variable]
cm:*physical-y-news-address-length* = $\log_2 q$	[Variable]
cm:*purely-virtual-cube-address-length* = $\log_2((v \cdot w)/(p \cdot q))$	[Variable]
cm:*purely-virtual-x-news-address-length* = $\log_2(v/p)$	[Variable]
cm:*purely-virtual-y-news-address-length* = $\log_2(w/q)$	[Variable]

These quantities are the base-two logarithms of the previously described quantities. They are useful for measuring the lengths (in bits) of fields containing processor numbers and memory addresses.

3.7 Implementation Restrictions

cm:*maximum-integer-length*	[Variable]
-----------------------------	------------

Because of implementation restrictions, a few operations on signed and unsigned integers cannot handle operands longer than the value of cm:*maximum-integer-length*. These operations are:

cm:isqrt	cm:unsigned-isqrt
cm:float	cm:unsigned-float
cm:*	cm:u*
cm:multiply	cm:unsigned-multiply
cm:floor-divide	cm:unsigned-floor-divide
cm:ceiling-divide	cm:unsigned-ceiling-divide
cm:truncate-divide	cm:unsigned-truncate-divide

cm:round-divide	cm:unsigned-round-divide
cm:mod	cm:unsigned-mod
cm:rem	cm:unsigned-rem
cm:floor-and-mod	cm:unsigned-floor-and-mod
cm:ceiling-and-remainder	cm:unsigned-ceiling-and-remainder
cm:truncate-and-rem	cm:unsigned-truncate-and-rem
cm:round-and-remainder	cm:unsigned-round-and-remainder
cm:get-from-north	cm:get-from-north-always
cm:get-from-east	cm:get-from-east-always
cm:get-from-west	cm:get-from-west-always
cm:get-from-south	cm:get-from-south-always

Experimentation might reveal that in certain cases some of these operations succeed when applied to operands that are longer than this variable, but that fact is not guaranteed in succeeding software releases.

The value of cm:*maximum-integer-length* is always at least as great as the smaller of 128 and half the value of cm:*user-memory-address-limit*.

cm:*maximum-significand-length*	[Variable]
cm:*maximum-exponent-length*	[Variable]

Because of implementation restrictions, a few operations on floating-point numbers cannot handle operands with significands or exponents longer than a certain size. These operations are cm:f+, cm:f-, cm:f*, cm:f/, and cm:float-sqrt.

Experimentation might reveal that in certain cases some of these operations succeed when applied to operands that are longer than specified by these variables, but that fact is not guaranteed in succeeding software releases.

The value of cm:*maximum-significand-length* is always at least as great as the smaller of 96 and eight less than half the value of cm:*user-memory-address-limit*.

The value of cm:*maximum-exponent-length* is always at least as great as the smaller of 32 and one-fourth the value of cm:*user-memory-address-limit*.

cm:*maximum-message-length*	[Variable]
-----------------------------	--------------

Because of implementation restrictions, a few operations that send messages from one processor to another cannot handle operands longer than a certain size. These operations are:

cm:send-with-overwrite	cm:store-with-overwrite
cm:send-with-logior	cm:store-with-logior
cm:send-with-logand	cm:store-with-logand
cm:send-with-logxor	cm:store-with-logxor
cm:send-with-add	cm:store-with-add
cm:send-with-max	cm:store-with-max
cm:send-with-min	cm:store-with-min
cm:send-with-unsigned-max	cm:store-with-unsigned-max

<code>cm:send-with-unsigned-min</code>	<code>cm:store-with-unsigned-min</code>
<code>cm:send</code>	<code>cm:store</code>
<code>cm:get</code>	<code>cm:fetch</code>
<code>cm:rank</code>	<code>unsigned-rank</code>
<code>cm:float-rank</code>	

All of these except the rank operations require the length of the data to be transmitted to be no greater than the value of `cm:*maximum-message-length*`. The rank operations have more complicated constraints; see their description for details.

Experimentation might reveal that in certain cases some of these operations succeed when applied to operands that are longer than specified by this variable, but that fact is not guaranteed in succeeding software releases.

The value of `cm:*maximum-message-length*` is always at least as great as the smaller of 192 and half the value of `cm:*user-memory-address-limit*`.

Chapter 4

Operation Formats and Addressing Modes

PARIS operations are executed at the direction of a program running in the front-end machine, typically a Digital Equipment Corporation VAX or a Symbolics 3600. For each operation there is a function or macro that, when called, causes the Connection Machine hardware to perform the operation.

The Lisp names of all the PARIS operations are in the `cm` package. To do a move operation, for example, one would write `(cm:move ...)` rather than `(move ...)`. This is particularly important to observe because many PARIS operations (intentionally) have the same name as commonly used Lisp operations.

4.1 Memory Addresses and the Stack

Most PARIS operations operate on one or more bit fields. A bit field may be a string of bits in the memory or may be one of the flags. The PARIS operations are described as if they were simple Lisp functions, but the user interface to many of the PARIS operations is actually a set of macros that accept a special syntax for operands that specify addresses of bit fields. Operands that are not bit fields have no special syntax, and any Lisp expression may be used to specify such operands.

In the following syntax description for bit-field-address operands, *x* is any Lisp expression.

<code>cm:test-flag</code>	The operand is the <code>test</code> flag.
<code>cm:carry-flag</code>	The operand is the <code>carry</code> flag.
<code>cm:overflow-flag</code>	The operand is the <code>overflow</code> flag.
<code>cm:context-flag</code>	The operand is the <code>context</code> flag. This will, in effect, always be 1 unless the operation is unconditional.
<code>(cm:stack x)</code>	The expression <i>x</i> must evaluate to a non-negative integer. The operand address is the stack pointer plus the value of <i>x</i> .
<i>x</i>	The expression <i>x</i> must evaluate to a non-negative integer. The operand address is the value of <i>x</i> .

Example: to add the 16-bit signed integer starting at location 48 to that starting at location 80, storing the result at location 80:

```
(cm:+ 80 48 16)
```

Example: to add 7 into the 16-bit signed integer starting at location 48:

```
(cm:+constant 48 7 16)
```

Example: to add the 13-bit field at location 72 to the 13-bit field on top of the stack:

```
(cm:+ (cm:stack 0) 72 13)
```

Example: to push a copy of the 13-bit field at location 51 onto the stack:

```
(cm:push-space 13)
(cm:move (cm:stack 0) 51 13)
```

Example: suppose there are three fields on the top of the stack whose (common) length is calculated at run time; their length is in the Lisp variable `truck-len`. To add the one that is third from the top to the one that is second from the top, leaving the topmost undisturbed:

```
(cm:+ (cm:stack truck-len) (cm:stack (* 2 truck-len)) truck-len)
```

Example: to pop a 13-bit field from the stack and store it at location 91:

```
(cm:move 91 (cm:stack 0) 13)
(cm:pop-and-discard 13)
```

4.2 Unconditional Operations

Most PARIS operations are conditional: they take place only in processors that have a 1 in the context flag. Sometimes it is necessary to perform operations unconditionally (that is, without respect to the context flag). Some operations have unconditional versions, generally named by appending `-always` to the name of the conditional function. For example, `cm:lognor-always` is the unconditional equivalent of `cm:lognor`. These operations are:

<code>cm:lognot-always</code>	<code>cm:move-always</code>
<code>cm:logand-always</code>	<code>cm:logior-always</code>
<code>cm:logxor-always</code>	<code>cm:logeqv-always</code>
<code>cm:lognand-always</code>	<code>cm:lognor-always</code>
<code>cm:logandc1-always</code>	<code>cm:logandc2-always</code>
<code>cm:logorc1-always</code>	<code>cm:logorc2-always</code>
<code>cm:get-from-north-always</code>	<code>cm:get-from-east-always</code>
<code>cm:get-from-west-always</code>	<code>cm:get-from-south-always</code>

A few PARIS operations have only unconditional forms and do not necessarily have names ending in `-always`.

Example: to unconditionally set the context flag to 1 in every processor:

```
(cm:move-constant-always cm:context-flag 1 1)
```

(Note that the conditional version

```
(cm:move-constant cm:context-flag 1 1)
```

has no net effect because it is executed only in those processors whose context flag is already 1.)

Example: to unconditionally copy the overflow flag into the context flag:

```
(cm:move-always cm:context-flag cm:overflow-flag 1)
```

Example: to unconditionally EXCLUSIVE OR the overflow flag into the context flag:

```
(cm:logxor-always cm:context-flag cm:overflow-flag 1)
```

Example: to unconditionally push the context flag onto the stack:

```
(cm:push-space 1)
(cm:move-always (cm:stack 0) cm:context-flag 1)
```

4.3 Constant Operands

Certain operations accept as an operand a single datum computed within the front end that is broadcast to all of the Connection Machine processors as part of the operation. These are:

cm:+constant	cm:u+constant
cm:-constant	cm:u-constant
cm:=constant	cm:u=constant
cm:/=constant	cm:u/=constant
cm:<constant	cm:u<constant
cm:<=constant	cm:u<=constant
cm:>constant	cm:u>constant
cm:>=constant	cm:u>=constant
cm:max-constant	cm:unsigned-max-constant
cm:min-constant	cm:unsigned-min-constant
cm:move-constant	cm:move-constant-always
cm:float-move-constant	cm:float-move-decoded-constant

For example, the second argument to the `cm:+constant` operation is a constant operand; `(cm:+constant 43 1 32)` will, in those processors that are selected, add the value 1 to the 32-bit field starting at location 43. Similarly, `(cm:<constant 43 10000 16)` will set the test flag in every selected processor in which the 16-bit field starting at location 43 contains a (signed) value that is less than ten thousand.

4.4 Flags as Operands

Certain operations allow the flags as operands. These are:

<code>cm:move</code>	<code>cm:move-always</code>
<code>cm:move-constant</code>	<code>cm:move-constant-always</code>
<code>cm:lognot</code>	<code>cm:lognot-always</code>
<code>cm:logand</code>	<code>cm:logand-always</code>
<code>cm:logior</code>	<code>cm:logior-always</code>
<code>cm:logxor</code>	<code>cm:logxor-always</code>
<code>cm:logeqv</code>	<code>cm:logeqv-always</code>
<code>cm:lognand</code>	<code>cm:lognand-always</code>
<code>cm:logandc1</code>	<code>cm:logandc1-always</code>
<code>cm:logandc2</code>	<code>cm:logandc2-always</code>
<code>cm:lognor</code>	<code>cm:lognor-always</code>
<code>cm:logorc1</code>	<code>cm:logorc1-always</code>
<code>cm:logorc2</code>	<code>cm:logorc2-always</code>
<code>cm:get-from-north</code>	<code>cm:get-from-north-always</code>
<code>cm:get-from-east</code>	<code>cm:get-from-east-always</code>
<code>cm:get-from-west</code>	<code>cm:get-from-west-always</code>
<code>cm:get-from-south</code>	<code>cm:get-from-south-always</code>
<code>cm:global-count</code>	<code>cm:global-count-always</code>
<code>cm:global-logand</code>	<code>cm:global-logand-always</code>
<code>cm:global-logior</code>	<code>cm:global-logior-always</code>
<code>cm:latch-leds</code>	<code>cm:latch-leds-always</code>
<code>cm:unsigned-read-from-processor</code>	<code>cm:unsigned-write-to-processor</code>

No other PARIS operation may legitimately be used to operate on the flags.

Example: to conditionally copy the overflow flag into the context flag:

```
(cm:move cm:context-flag cm:overflow-flag 1)
```

Example: to count all the selected processors that have the test flag set:

```
(setq the-count (cm:global-count cm:test-flag))
```

(The `cm:global-count` operation returns the count as a Lisp value.)

Chapter 5

Getting Started

Before PARIS operations can be used to control a Connection Machine system, three things must be done:

- First, the PARIS software must be loaded into the front-end computer execution environment. The procedures for doing this are dependent on both the programming language and the installation, and are described in the *Connection Machine System User's Guide*.
- Second, Connection Machine processors must be allocated for use. This is done with the `cm:attach` operation. If actual Connection Machine hardware is to be used, `cm:attach` allocates physical processors and appropriately configures them as virtual processors. If the PARIS simulator is to be used, `cm:attach` configures the simulator to simulate the specified number of virtual processors.
- Third, the Connection Machine processors and memory must be initialized for use. The `cm:cold-boot` operation does this.

Typically the PARIS software need be loaded into a Lisp environment only once. It is possible to attach processors for use, release them (using the `cm:detach` operation), and then later attach processors again (possibly the same processors or different ones). Detaching processors makes them available for use by other front-end computers, but of course their state and memory contents are then lost.

While processors remain attached it may be desirable to reinitialize the state of the processors. One may use `cm:cold-boot` at any time to reset the processors to a standard initial state. It is a good idea for an application to call `cm:cold-boot` before doing anything else.

There is also an operation `cm:warm-boot` to reset the processors while preserving the contents of memory. If a computation using the Connection Machine system should be interrupted at some point and not continued, it is usually best to call either `cm:cold-boot` or `cm:warm-boot` before invoking any other PARIS operation.

Here is a transcript of a short session on a Symbolics 3600 illustrating the use of the Connection Machine system. In practice, of course, one does not always type in PARIS instructions one at a time from the keyboard, but writes large programs that call PARIS

operations. This sample session does demonstrate interactive debugging in a Lisp-based system, as well as illustrate the use of `cm:attach`, `cm:cold-boot`, and `cm:detach`.

```
(setup-cm-environment)
```

This causes a menu to pop up; using the mouse, the user (call him George) chooses to use PARIS with hardware. Now hardware must be allocated.

```
(cm:attach :16k)
```

```
NIL
```

Now George is attached to 16,384 physical processors.

```
(cm:finger)
```

Connection Machine System Rainbow		Physical size: 64K processors with 4K RAM	
Jane	Red	Microcontroller Port (3)	<-- 16384 physical processors
Judy	Blue	Not Attached to a Port	
George	Yellow	Microcontroller Port (1)	<-- 16384 physical processors
Elroy	Green	Microcontroller Port (0)	<-- 16384 physical processors

From what `cm:finger` prints we can see that George is using the front-end computer named Yellow, that the Connection Machine system is named Rainbow and has a total of 65,536 physical processors, and that two other users, Jane and Elroy, are also attached to portions of the Connection Machine system. User Judy is logged in to the front-end computer named Blue, but is not attached to any Connection Machine hardware.

```
(cm:cold-boot)
```

```
16384
```

```
16384
```

```
3579
```

George cold-boots his portion of the Connection Machine system. He allows the number of virtual processors to default to the number of physical processors. There are 3,579 bits of memory per virtual processor available to the user.

```
(cm:my-cube-address 0)
```

```
NIL
```

```
(cm:global-add 0 cm:*cube-address-length*)
```

```
(error)
```

George causes every processor to put its own cube address into memory starting at location 0. He then tries to calculate the sum of these addresses (thereby summing the integers from 0 to 16,383) but an error occurs. (The details of the error message are system-dependent and are not shown here.) The problem is that the `cm:global-add` operation requires the use of temporary scratch space in the gap, and no gap space has been allocated since the `cm:cold-boot` operation, which resets the stack, gap, and upper data areas to be empty.


```
(cm:set-stack-limit (- (cm:get-stack-limit) 50))
3528
```

```
(cm:global-add 0 cm:*cube-address-length*)
-8192
```

George allocates 50 bits of gap area and tries again to take the sum. The result is surprising: a negative number. Aha! Cube addresses should be treated as unsigned integers; treating them as signed causes half of them to be considered negative.

```
(cm:global-unsigned-add 0 cm:*cube-address-length*)
134209536
```

Using an unsigned summation produces a more reasonable result. But is it correct?

```
(/ (* 16383 16384) 2)
134209536
```

Yes, that is the right answer.

```
(cm:cold-boot)
16384
16384
3579
```

```
(cm:global-unsigned-add 0 cm:*cube-address-length*)
136
```

Calling `cm:cold-boot` can cause the contents of memory to be lost.

```
(cm:my-cube-address 0)
NIL
```

```
(cm:global-add 0 cm:*cube-address-length*)
134209536
```

After reconstructing the cube address values, George then gets the correct answer again for the sum.

At this point Jane politely requests that George and Elroy detach their processors so that she can attach all 65,536 processors for a production run of her application.

```
(cm:detach)
NIL
```

George releases the processors he had attached so that Jane can use them.

```
(cm:global-unsigned-add 0 cm:*cube-address-length*)
(error)
```

George tries to take the sum again, just to see what will happen. An error is signalled, indicating that he no longer has hardware attached.

Chapter 6

Simple Unary Operations

Each function in this chapter operates on one field and produces some result. The operation is performed in every processor selected by the context flag. In most cases some function of a *source* field is computed and stored in a *destination* field of the same length. The two fields are specified by three arguments: the *destination* address, the *source* address, and the common *length* of the two fields. (Sometimes the name *destination* is abbreviated to *dest*.)

For some operations, such as sign tests, the only effect is to set flags, and there is no destination field as such.

For every operation in this section, the same field may be used for both operands to a single operation, and the operation will behave in the expected manner. Two fields are the same only if they are specified by identical argument values and also are treated by the operation as having the same data type. The results are unpredictable if the two operand fields overlap but are not identical.

If a flag is not explicitly mentioned in the individual description of an operation, then the operation leaves the flag unaffected.

6.1 Unary Operations on Bit Fields

<code>cm:lognot destination source length</code>	[Operation]
<code>cm:lognot-always destination source length</code>	[Operation]

Each bit of the *destination* field is set to the logical NOT of the corresponding bit of the *source* field.

`cm:lognot-always` is an unconditional version of `cm:lognot`.

6.2 Unary Operations on Signed Integers

<code>cm:abs destination source length</code>	[Operation]
---	-------------

The absolute value of the source is placed in the destination.

The **overflow** flag is set if the result cannot be represented in the destination field, and is cleared otherwise. Overflow occurs only when the source value is the most negative representable number.

cm:negate *destination source length* [Operation]

The negative of the source is placed in the destination.

The **overflow** flag is set if the result cannot be represented in the destination field, and is cleared otherwise. Overflow occurs only when the source value is the most negative representable number.

cm:signum *destination source dlen slen* [Operation]

The signed integer value -1, 0, or 1 is placed in the destination according to whether the source value is negative, zero, or positive, respectively. The destination field need not be the same length as the source field; 2 is a popular length.

cm:isqrt *destination source length* [Operation]

The integer square root of the source value is placed in the destination; this is the largest integer not greater than the mathematical square root.

If the source value is negative, then the **test** flag is set and the destination value is unpredictable; otherwise the **test** flag is cleared.

The *length* must be not greater than the value of **cm:*maximum-integer-length***.

cm:zerop *source length* [Operation]

The **test** flag is set if the *source* field is zero, and is cleared otherwise.

cm:minusp *source length* [Operation]

The **test** flag is set if the *source* field is negative, and is cleared otherwise.

cm:plusp *source length* [Operation]

The **test** flag is set if the *source* field is positive, and is cleared otherwise.

cm:float *dest source slen &optional (signif-len 23) (expt-len 8)* [Operation]

The *source* field, treated as a signed integer, is converted to a floating-point number and placed into the destination. The length of the *source* field is specified by *slen*; the format of the *dest* field is specified by *signif-len* and *expt-len*.

The **overflow** flag is set if the source value cannot be represented in the destination field, and is unaffected otherwise.

The length *slen* must be not greater than the value of **cm:*maximum-integer-length***.

cm:new-size *destination source dlen slen* [Operation]

The *source* field, treated as a signed integer, is copied into the *destination* field.

If the length of the destination is equal to the length of the source, then the overflow flag is cleared.

If the length of the destination is greater than the length of the source, then the source field is sign-extended, and the overflow flag is cleared.

If the length of the destination is less than the length of the source, then overflow checking is performed. The overflow flag is set if the source value cannot be represented in the destination field, and is cleared otherwise.

cm:integer-length *destination source dlen slen* [Operation]

The destination receives, as an *unsigned* integer, the result of the computation

$$\begin{array}{ll} \lceil \log_2(s + 1) \rceil & \text{if } s \geq 0 \\ \lceil \log_2(-s) \rceil & \text{if } s < 0 \end{array}$$

where s is the source value. This quantity is one less than the minimum number of bits required to represent s as a signed number.

The overflow flag is set if the result cannot be represented in the destination field, and is cleared otherwise.

cm:logcount *destination source dlen slen* [Operation]

The destination receives, as an *unsigned* integer, a count of the number of bits in the representation of the source value that are different from the sign bit.

The overflow flag is set if the result cannot be represented in the destination field, and is cleared otherwise.

6.3 Unary Operations on Unsigned Integers

cm:unsigned-negate *destination source length* [Operation]

The “unsigned negative” of the source (that is, the unsigned result of subtracting the source from zero) is placed in the destination.

The overflow flag is set if the source value is non-zero, and is cleared otherwise.

cm:unsigned-isqrt *destination source length* [Operation]

The integer square root of the unsigned source value is placed in the destination; this is the largest integer not greater than the mathematical square root.

The *length* must be not greater than the value of **cm:*maximum-integer-length***.

cm:unsigned-zerop *source length* [Operation]

The **test** flag is set if the *source* field is zero, and is cleared otherwise. Of course, **unsigned-zerop** behaves identically to **zerop**, and is provided primarily for reasons of symmetry.

cm:unsigned-plusp *source length* [Operation]

The **test** flag is set if the *source* field is non-zero, and is cleared otherwise.

cm:unsigned-float *dest source slen* [Operation]
 &optional (*signif-len* 23) (*expt-len* 8)

The *source* field, treated as an unsigned integer, is converted to a floating-point number and placed into the destination. The length of the *source* field is specified by *slen*; the format of the *dest* field is specified by *signif-len* and *expt-len*.

The overflow flag is set if the source value cannot be represented in the destination field, and is unaffected otherwise.

The length *slen* must be not greater than the value of *cm:*maximum-integer-length**.

cm:unsigned-new-size *destination source dlen slen* [Operation]

The *source* field, treated as an unsigned integer, is copied into the *destination* field.

If the length of the destination is equal to the length of the source, then the overflow flag is cleared.

If the length of the destination is greater than the length of the source, then the source field is zero-extended, and the overflow flag is cleared.

If the length of the destination is less than the length of the source, then overflow checking is performed. The overflow flag is set if the source value cannot be represented in the destination field, and is cleared otherwise.

cm:unsigned-integer-length *destination source dlen slen* [Operation]

The destination receives, as an unsigned integer, the value $\lceil \log_2(s + 1) \rceil$ where *s* is the source value. This quantity is the minimum number of bits required to represent *s* as an unsigned number.

The overflow flag is set if the source value cannot be represented in the destination field, and is cleared otherwise.

cm:unsigned-logcount *destination source dlen slen* [Operation]

The destination receives, as an unsigned integer, a count of the number of 1-bits in the representation of the source value.

The overflow flag is set if the source value cannot be represented in the destination field, and is cleared otherwise.

cm:gray-code-from-integer *destination source length* [Operation]

The source value is treated as an unsigned integer and converted to a Gray code representation, which is stored in the destination.

cm:front-end-gray-code-from-integer *integer* [Function]

This operation is performed entirely within the front-end computer. The argument should be a non-negative integer; it is converted to a Gray code representation, which is returned.

cm:integer-from-gray-code *destination source length* [Operation]

The source value is treated as a Gray code value and converted to an unsigned integer, which is stored in the destination.

cm:front-end-integer-from-gray-code *gray-code* [Function]

This operation is performed entirely within the front-end computer. The argument should be a non-negative integer; it is treated as a Gray code value and converted to a non-negative integer, which is returned.

6.4 Unary Operations on Floating-Point Numbers

cm:float-abs *dest source &optional (signif-len 23) (expt-len 8)* [Operation]

The absolute value of the source is placed in the destination.

cm:float-negate *dest source &optional (signif-len 23) (expt-len 8)* [Operation]

The negative of the source is placed in the destination.

cm:float-signum *dest source dlen &optional (signif-len 23) (expt-len 8)* [Operation]

The signed integer value -1, 0, or 1 is placed in the destination according to whether the source value is negative, zero, or positive, respectively. The length of the *dest* field is specified by *dlen*; the format of the *source* field is specified by *signif-len* and *expt-len*. The length of the destination field must be at least 2.

cm:float-float-signum *dest source &optional (signif-len 23) (expt-len 8)* [Operation]

The floating-point value -1.0, -0.0, +0.0, or 1.0 is placed in the destination according to whether the source value is negative, minus zero, plus zero, or positive, respectively.

cm:float-sqrt *dest source &optional (signif-len 23) (expt-len 8)* [Operation]

The square root of the source value is placed in the destination.

Note that the length *signif-len* must be not greater than the value of *cm:*maximum-significand-length**, and the length *expt-len* must be not greater than the value of *cm:*maximum-exponent-length**.

If the source value is negative (other than minus zero), then the test flag is set and the square root of the absolute value of the source is calculated. If the source value is minus zero, then test flag is set and the result is minus zero. In all other cases the test flag is cleared.

cm:float-zerop *source &optional (signif-len 23) (expt-len 8)* [Operation]

The test flag is set if the *source* field is zero (plus or minus), and is cleared otherwise.

cm:float-minusp *source* &optional (*signif-len* 23) (*expt-len* 8) [Operation]

The test flag is set if the *source* field is negative (but not minus zero), and is cleared otherwise.

cm:float-plusp *source* &optional (*signif-len* 23) (*expt-len* 8) [Operation]

The test flag is set if the *source* field is positive (but not plus zero), and is cleared otherwise.

cm:floor *dest source dlen* &optional (*signif-len* 23) (*expt-len* 8) [Operation]

cm:ceiling *dest source dlen* &optional (*signif-len* 23) (*expt-len* 8) [Operation]

cm:truncate *dest source dlen* [Operation]
&optional (*signif-len* 23) (*expt-len* 8)

cm:round *dest source dlen* &optional (*signif-len* 23) (*expt-len* 8) [Operation]

The *source* field, treated as a floating-point number, is converted to a signed integer and placed into the destination. The length of the *dest* field is specified by *dlen*; the format of the *source* field is specified by *signif-len* and *expt-len*.

The four operations differ only in the method of rounding when the source value is not an exact integer:

floor rounds toward $-\infty$.

ceiling rounds toward $+\infty$.

truncate rounds toward 0.

round rounds to the nearest integer, and if the source value lies exactly halfway between two integers, then the even integer is used.

The overflow flag is set if the source value cannot be represented in the destination field, and is cleared otherwise.

cm:unsigned-floor *dest source dlen* [Operation]
&optional (*signif-len* 23) (*expt-len* 8)

cm:unsigned-ceiling *dest source dlen* [Operation]
&optional (*signif-len* 23) (*expt-len* 8)

cm:unsigned-truncate *dest source dlen* [Operation]
&optional (*signif-len* 23) (*expt-len* 8)

cm:unsigned-round *dest source dlen* [Operation]
&optional (*signif-len* 23) (*expt-len* 8)

These are identical to the signed versions described above, but produce unsigned integer results.

The overflow flag is set if the source value cannot be represented in the destination field, and is cleared otherwise. Minus zero produces a zero result and does not set the overflow flag.

cm:float-new-size *dest source d-signif-len d-expt-len s-signif-len* [Operation]
s-expt-len

The *source* field, treated as a floating-point number, is copied into the *destination* field

in a new format. The format of the *dest* field is specified by *d-signif-len* and *d-expt-len*; the format of the *source* field is specified by *s-signif-len* and *s-expt-len*.

If the significand field of the destination is less than that of the source, then rounding is performed. This may lead to overflow.

If the exponent field of the destination is less than that of the source, then overflow may occur.

The overflow flag is set if the source value cannot be represented in the destination field, and is unaffected otherwise.

Chapter 7

Simple Binary Operations

Each function in this chapter operates on two fields. The operation is performed in every processor selected by the context flag. In most cases both fields are the same length, and a single *length* operand is provided.

Typically one operand field receives the result and also serves as the first input; it is called the *destination* even though it is also one source. The other operand field is called the *source*. The two fields are specified by three arguments *destination*, *source*, and the common *length* of the two fields. (Sometimes the name *destination* is abbreviated to *dest*.)

For some operations, such as relational comparisons, the only effect is to set flags, and there is no destination field as such. In that case the two fields are specified by arguments called *source1*, *source2*, and *length*.

For every operation in this section, the same field may be used for both operands to a single operation, and the operation will behave in the expected manner. The results are unpredictable if the two operand fields overlap but are not identical.

If a flag is not explicitly mentioned in the individual description of an operation, then the operation leaves the flag unaffected.

7.1 Binary Operations on Bit Fields

All ten non-trivial boolean operations of two arguments are provided, in both conditional and unconditional versions.

<code>cm:logand</code>	<i>destination source length</i>	[<i>Operation</i>]
<code>cm:logand-always</code>	<i>destination source length</i>	[<i>Operation</i>]

Each bit of the *destination* field is set to the logical AND of that bit and the corresponding bit of the *source* field. The `cm:logand` operation is conditional, and `cm:logand-always` is unconditional.

These operations may be used to operate on the flags.

`cm:logior destination source length` [Operation]

`cm:logior-always destination source length` [Operation]

Each bit of the *destination* field is set to the logical INCLUSIVE OR of that bit and the corresponding bit of the *source* field. The `cm:logior` operation is conditional, and `cm:logior-always` is unconditional.

These operations may be used to operate on the flags.

`cm:logxor destination source length` [Operation]

`cm:logxor-always destination source length` [Operation]

Each bit of the *destination* field is set to the logical EXCLUSIVE OR of that bit and the corresponding bit of the *source* field. The `cm:logxor` operation is conditional, and `cm:logxor-always` is unconditional.

These operations may be used to operate on the flags.

`cm:logeqv destination source length` [Operation]

`cm:logeqv-always destination source length` [Operation]

Each bit of the *destination* field is set to the logical EQUIVALENCE (also called EXCLUSIVE NOR) of that bit and the corresponding bit of the *source* field. The `cm:logeqv` operation is conditional, and `cm:logeqv-always` is unconditional.

These operations may be used to operate on the flags.

`cm:lognand destination source length` [Operation]

`cm:lognand-always destination source length` [Operation]

Each bit of the *destination* field is set to the logical NAND (that is, the NOT of the AND) of that bit and the corresponding bit of the *source* field. The `cm:lognand` operation is conditional, and `cm:lognand-always` is unconditional.

These operations may be used to operate on the flags.

`cm:lognor destination source length` [Operation]

`cm:lognor-always destination source length` [Operation]

Each bit of the *destination* field is set to the logical OR (that is, the NOT of the OR) of that bit and the corresponding bit of the *source* field. The `cm:lognor` operation is conditional, and `cm:lognor-always` is unconditional.

These operations may be used to operate on the flags.

`cm:logandc1 destination source length` [Operation]

`cm:logandc1-always destination source length` [Operation]

Each bit of the *destination* field is set to the logical AND of (1) the NOT of that bit, and (2) the corresponding bit of the *source* field. The `cm:logandc1` operation is conditional, and `cm:logandc1-always` is unconditional.

These operations may be used to operate on the flags.

cm:logandc2 destination source length [Operation]

cm:logandc2-always destination source length [Operation]

Each bit of the *destination* field is set to the logical AND of (1) that bit, and (2) the NOT of the corresponding bit of the *source* field. The *cm:logandc2* operation is conditional, and *cm:logandc2-always* is unconditional.

These operations may be used to operate on the flags.

cm:logorc1 destination source length [Operation]

cm:logorc1-always destination source length [Operation]

Each bit of the *destination* field is set to the logical OR of (1) the NOT of that bit, and (2) the corresponding bit of the *source* field. The *cm:logorc1* operation is conditional, and *cm:logorc1-always* is unconditional.

These operations may be used to operate on the flags.

cm:logorc2 destination source length [Operation]

cm:logorc2-always destination source length [Operation]

Each bit of the *destination* field is set to the logical OR of (1) that bit, and (2) the NOT of the corresponding bit of the *source* field. The *cm:logorc2* operation is conditional, and *cm:logorc2-always* is unconditional.

These operations may be used to operate on the flags.

7.2 Binary Operations on Signed Integers

cm:+ destination source length [Operation]

cm:+constant destination source-constant length [Operation]

For *cm:+*, the source is added into the destination. The source and destination fields must be of the same length.

For *cm:+constant*, the source-constant (a Lisp integer) is added into the destination.

The carry flag receives the carry out from the high-order bit position. The length must be greater than or equal to one.

The overflow flag is set if the sum cannot be represented in the destination field, and is cleared otherwise; it is computed as the XOR of the carry into the high-order bit position and the carry out of the high-order bit position.

cm:+carry destination source length [Operation]

The source and the carry flag are together added into the destination; that is, the destination receives the three-way sum of the destination, the source, and the old carry flag. The source and destination fields must be of the same length. The carry flag then receives the carry out from the high-order bit position. The length must be greater than or equal to one.

The overflow flag is set if the sum cannot be represented in the destination field, and is cleared otherwise; it is computed as the XOR of the carry into the high-order bit position and the carry out of the high-order bit position.

cm:+flags source1 source2 length [Operation]

The two sources are added together. The source fields must be of the same length. The length must be greater than or equal to one.

The carry flag and overflow flag are set as for the *cm:+* operation, but the sum itself is not written into memory. Only the flags are affected.

cm:add dest source1 source2 dlen slen1 slen2 [Operation]

The two source fields are added together and the sum is stored in the destination. The length of the *dest* field is specified by *dlen*; the length of the *source1* field is specified by *slen1*; the length of the *source2* field is specified by *slen2*. The lengths must be greater than or equal to one.

The overflow flag is set if the sum cannot be represented in the destination field, and is cleared otherwise.

If you wish to use only two addresses, *dest* may point to the same field as *source1*. No other overlapping combination is correct.

This operation differs from the *cm:+* operation only in allowing independent specification of the destination and first source, in allowing independent specification of the lengths of all three fields, and in not affecting the carry flag.

cm:- destination source length [Operation]

cm:-constant destination source-constant length [Operation]

For *cm:-*, the source is subtracted from the destination. The source and destination fields must be of the same length.

For *cm:-constant*, the source-constant (a Lisp integer) is subtracted from the destination.

The carry flag then receives the carry out from the high-order bit position; "carry" is equivalent to "not borrow." The length must be greater than or equal to one.

The overflow flag is set if the difference cannot be represented in the destination field, and is cleared otherwise.

cm:-borrow destination source length [Operation]

The source and the inverse of the carry flag are together subtracted from the destination. The carry flag then receives the carry out from the high-order bit position; "carry" is equivalent to "not borrow." The length must be greater than or equal to one.

The overflow flag is set if the difference cannot be represented in the destination field, and is cleared otherwise.

cm:subtract dest source1 source2 dlen slen1 slen2 [Operation]

The second source field is subtracted from the first source field, and the difference is stored in the destination. The length of the *dest* field is specified by *dlen*; the length of the *source1* field is specified by *slen1*; the length of the *source2* field is specified by *slen2*. The lengths must be greater than or equal to one.

The overflow flag is set if the sum cannot be represented in the destination field, and is cleared otherwise.

This operation differs from the `cm:-` operation only in allowing independent specification of the destination and first source, in allowing independent specification of the lengths of all three fields, and in not affecting the carry flag.

`cm:* destination source length` [Operation]

The source is multiplied by the destination. The overflow flag is set if the product cannot be represented in the destination field, and is cleared otherwise.

The *length* must be not greater than the value of `cm:*maximum-integer-length*`.

`cm:multiply dest source1 source2 dlen slen1 slen2` [Operation]

The two source fields are multiplied together and the product is stored in the destination. The length of the *dest* field is specified by *dlen*; the length of the *source1* field is specified by *slen1*; the length of the *source2* field is specified by *slen2*.

The overflow flag is set if the product cannot be represented in the destination field, and is cleared otherwise.

The lengths *slen1* and *slen2* must each be not greater than the value of `cm:*maximum-integer-length*`.

This operation differs from the `*` operation only in allowing independent specification of the destination and first source and independent specification of the lengths of all three fields.

<code>cm:floor-divide dest source1 source2 dlen slen1 slen2</code>	[Operation]
<code>cm:ceiling-divide dest source1 source2 dlen slen1 slen2</code>	[Operation]
<code>cm:truncate-divide dest source1 source2 dlen slen1 slen2</code>	[Operation]
<code>cm:round-divide dest source1 source2 dlen slen1 slen2</code>	[Operation]

The first source field is divided by the second source field, and the quotient is stored in the destination. The four operations differ only in the method of rounding when the true mathematical quotient is not an exact integer:

`floor-divide` rounds toward $-\infty$.

`ceiling-divide` rounds toward $+\infty$.

`truncate-divide` rounds toward 0.

`round-divide` rounds to the nearest integer, and if the mathematical quotient lies exactly halfway between two integers, then the even integer is used.

The length of the *dest* field is specified by *dlen*; the length of the *source1* field is specified by *slen1*; the length of the *source2* field is specified by *slen2*. The lengths *slen1* and *slen2* must each be not greater than the value of `cm:*maximum-integer-length*`.

The overflow flag is set if the quotient cannot be represented in the destination field, and is cleared otherwise. If the second source is zero, then the test flag is set and the destination field will contain unpredictable results; otherwise the test flag is cleared.

greater (*cm:max-constant*) or less (*cm:min-constant*) than the destination, then the source-constant is copied to the destination and the **test** flag is set; otherwise the **test** flag is cleared and the destination is unchanged.

<i>cm:= source1 source2 length</i>	[Operation]
<i>cm:/= source1 source2 length</i>	[Operation]
<i>cm:< source1 source2 length</i>	[Operation]
<i>cm:<= source1 source2 length</i>	[Operation]
<i>cm:> source1 source2 length</i>	[Operation]
<i>cm:>= source1 source2 length</i>	[Operation]

The *source1* field is compared with the *source2* field, treating them as signed integers. The **test** flag is set if the *source1* field is equal to (*=*), not equal to (*/=*), less than (*<*), less than or equal to (*<=*), greater than (*>*), or greater than or equal to (*>=*) the *source2* field, and is cleared otherwise.

Note that in COMMON LISP the not-equal operation may be written simply as */=* but when using Zetalisp on the Symbolics 3600 the notation *//=* must be used because */* is an escape character in Zetalisp.

<i>cm:=constant source source-constant length</i>	[Operation]
<i>cm:/=constant source source-constant length</i>	[Operation]
<i>cm:<constant source source-constant length</i>	[Operation]
<i>cm:<=constant source source-constant length</i>	[Operation]
<i>cm:>constant source source-constant length</i>	[Operation]
<i>cm:>=constant source source-constant length</i>	[Operation]

The source is compared with the source-constant, treating them as signed integers. The **test** flag is set if the *source* field is equal to (*=*), not equal to (*/=*), less than (*<*), less than or equal to (*<=*), greater than (*>*), or greater than or equal to (*>=*) the *source-constant* (a Lisp integer), and is cleared otherwise.

Note that in COMMON LISP the not-equal operation may be written simply as */=constant* but when using Zetalisp on the Symbolics 3600 the notation *//=constant* must be used because */* is an escape character in Zetalisp.

<i>cm:compare dest source1 source2 dlen slen1 slen2</i>	[Operation]
---	-------------

The two source fields are compared, and the value -1, 0, or 1 is stored in the destination according to whether the *source1* field is less than, equal to, or greater than the *source2* field, respectively. The length of the destination field must be at least 2.

The length of the *destination* field is specified by *dlen*; the length of the *source1* field is specified by *slen1*; the length of the *source2* field is specified by *slen2*.

<i>cm:shift destination source dlen slen</i>	[Operation]
--	-------------

The contents of the *destination* field are shifted within that field by a distance determined by the *source* field (a signed integer). In this way the *destination* field can be shifted by a different amount in each selected processor.

If the *source* field is positive, the shift is to the left (toward the most significant end of the field), and vacated low-order positions are cleared. The *overflow* flag is set if any bit that differs from the sign bit is shifted into or past the sign bit, and otherwise is cleared.

If the *source* field is negative, the shift is to the right (toward the least significant end of the field), and vacated high-order positions are filled with copies of the original sign bit of the *destination* field. The *overflow* flag is always cleared in this case.

If the *source* field is zero, then the *destination* field is unaffected. The *overflow* flag is always cleared in this case.

7.3 Binary Operations on Unsigned Integers

The operations in this section parallel exactly those in the previous section, but operate on unsigned integers rather than signed integers. The descriptions do differ in minor ways, such as in observations about overflow, because of the difference in representation.

cm:u+ destination source length [Operation]
cm:u+constant destination source-constant length [Operation]

For *cm:u+*, the source is added into the destination. The source and destination fields must be of the same length.

For *cm:u+constant*, the source-constant (a non-negative Lisp integer) is added into the destination.

The *carry* flag receives the carry out from the high-order bit position. The length must be greater than or equal to one.

The *overflow* flag is set if the sum cannot be represented in the destination field, and is cleared otherwise; it is computed as the carry out of the high-order bit position.

cm:u+carry destination source length [Operation]

The source and the *carry* flag are together added into the destination; that is, the destination receives the three-way sum of the destination, the source, and the old *carry* flag. The source and destination fields must be of the same length. The *carry* flag then receives the carry out from the high-order bit position. The length must be greater than or equal to one.

The *overflow* flag is set if the sum cannot be represented in the destination field, and is cleared otherwise; it is computed as the carry out of the high-order bit position.

cm:u+flags source1 source2 length [Operation]

The two sources are added together as unsigned integers. The source fields must be of the same length. The length must be greater than or equal to one.

The *carry* flag and *overflow* flag are set as for the *cm:u+* operation, but the sum itself is not written into memory. Only the flags are affected.

cm:unsigned-add dest source1 source2 dlen slen1 slen2 [Operation]

The two source fields are added together and the sum is stored in the destination. The length of the *dest* field is specified by *dlen*; the length of the *source1* field is specified by *slen1*; the length of the *source2* field is specified by *slen2*. The lengths must be greater than or equal to one.

The overflow flag is set if the sum cannot be represented in the destination field, and is cleared otherwise.

This operation differs from the *u+* operation only in allowing independent specification of the destination and first source, in allowing independent specification of the lengths of all three fields, and in not affecting the carry flag.

cm:u- destination source length [Operation]

cm:u-constant destination source-constant length [Operation]

For *cm:u-*, the source is subtracted from the destination. The source and destination fields must be of the same length.

For *cm:u-constant*, the source-constant (a non-negative Lisp integer) is subtracted from the destination.

The carry flag then receives the carry out from the high-order bit position; "carry" is equivalent to "not borrow." The length must be greater than or equal to one.

The overflow flag is set if the difference cannot be represented in the destination field, and is cleared otherwise.

cm:u-borrow destination source length [Operation]

The source and the inverse of the carry flag are together subtracted from the destination. The carry flag then receives the carry out from the high-order bit position; "carry" is equivalent to "not borrow." The length must be greater than or equal to one.

The overflow flag is set if the difference cannot be represented in the destination field, and is cleared otherwise.

cm:unsigned-subtract dest source1 source2 dlen slen1 slen2 [Operation]

The second source field is subtracted from the first source field, and the difference is stored in the destination. The length of the *dest* field is specified by *dlen*; the length of the *source1* field is specified by *slen1*; the length of the *source2* field is specified by *slen2*. The lengths must be greater than or equal to one.

The overflow flag is set if the sum cannot be represented in the destination field, and is cleared otherwise.

This operation differs from the *u-* operation only in allowing independent specification of the destination and first source, in allowing independent specification of the lengths of all three fields, and in not affecting the carry flag.

cm:u destination source length* [Operation]

The source is multiplied by the destination. The overflow flag is set if the product cannot be represented in the destination field, and is cleared otherwise.

`cm:unsigned-multiply dest source1 source2 dlen slen1 slen2` [Operation]

The two source fields are multiplied together and the product is stored in the destination. The length of the *dest* field is specified by *dlen*; the length of the *source1* field is specified by *slen1*; the length of the *source2* field is specified by *slen2*.

The overflow flag is set if the product cannot be represented in the destination field, and is cleared otherwise.

This operation differs from the `*` operation only in allowing independent specification of the destination and first source and independent specification of the lengths of all three fields.

`cm:unsigned-floor-divide dest source1 source2 dlen slen1 slen2` [Operation]

`cm:unsigned-ceiling-divide dest source1 source2 dlen slen1 slen2` [Operation]

`cm:unsigned-truncate-divide dest source1 source2 dlen slen1 slen2` [Operation]

`cm:unsigned-round-divide dest source1 source2 dlen slen1 slen2` [Operation]

The first source field is divided by the second source field, and the quotient is stored in the destination. The four operations differ only in the method of rounding when the true mathematical quotient is not an exact integer:

`unsigned-floor-divide` rounds toward $-\infty$.

`unsigned-ceiling-divide` rounds toward $+\infty$.

`unsigned-truncate-divide` rounds toward 0.

`unsigned-round-divide` rounds to the nearest integer, and if the mathematical quotient lies exactly halfway between two integers, then the even integer is used.

Note that `unsigned-floor-divide` and `unsigned-truncate-divide` have identical behavior because the operands are unsigned and therefore always positive.

The length of the *dest* field is specified by *dlen*; the length of the *source1* field is specified by *slen1*; the length of the *source2* field is specified by *slen2*. The lengths *slen1* and *slen2* must each be not greater than the value of `cm:*maximum-integer-length*`.

The overflow flag is set if the quotient cannot be represented in the destination field, and is cleared otherwise. If the second source is zero, then the test flag is set and the destination field will contain unpredictable results; otherwise the test flag is cleared.

`cm:unsigned-mod destination source length` [Operation]

`cm:unsigned-rem destination source length` [Operation]

The destination field (call its value x) is divided by the source field (y), producing a quotient (q) and a remainder (r); the quotient is discarded, and the remainder is stored back in the *destination* field.

For `unsigned-mod`, the quotient is computed the same way as for `unsigned-floor-divide`; `unsigned-rem` makes the same computation as for `unsigned-truncate-divide`. The remainder is then always computed so as to satisfy the equation $x = q \cdot y + r$.

Because the operands are unsigned numbers, `unsigned-mod` and `unsigned-rem` have identical behavior.

The *length* must be not greater than the value of `cm:*maximum-integer-length*`.

Note that overflow cannot occur. If the second source is zero, then the **test** flag is set and the destination field will contain unpredictable results; otherwise the **test** flag is cleared.

cm:unsigned-floor-and-mod *dest1 dest2 source1 source2 dlen slen1 slen2* [Operation]
 cm:unsigned-ceiling-and-remainder *dest1 dest2 source1 source2 dlen slen1 slen2* [Operation]
 cm:unsigned-truncate-and-rem *dest1 dest2 source1 source2 dlen slen1 slen2* [Operation]
 cm:unsigned-round-and-remainder *dest1 dest2 source1 source2 dlen slen1 slen2* [Operation]

The first source field (call its value x) is divided by the second source field (y); the quotient (q) is stored in the *dest1* field, and the remainder (r) is stored in the *dest2* field. The quotient is computed the same way as for **unsigned-floor-divide**, **unsigned-ceiling-divide**, **unsigned-truncate-divide**, or **unsigned-round-divide**, respectively. The remainder is then always computed so as to satisfy the equation $x = q \cdot y + r$.

The length of the *dest1* field is specified by *dlen*; the length of the *source1* field is specified by *slen1*; the length of the *dest2* and *source2* fields is specified by *slen2*. The lengths *slen1* and *slen2* must each be not greater than the value of **cm:*maximum-integer-length***.

Because the operands are unsigned numbers, the operations **unsigned-floor-and-mod** and **unsigned-truncate-and-rem** have identical behavior.

The **overflow** flag is set if the quotient cannot be represented in the *dest1* field, and is cleared otherwise. Because the *dest2* always has the same size as the *source2* field, the remainder cannot overflow the *dest2* field. If the second source is zero, then the **test** flag is set and the destination fields will contain unpredictable results; otherwise the **test** flag is cleared.

cm:unsigned-max *destination source length* [Operation]
 cm:unsigned-min *destination source length* [Operation]

The source and destination fields are considered to be unsigned integers. If the source is greater (**cm:unsigned-max**) or less (**cm:unsigned-min**) than the destination, then the source is copied to the destination and the **test** flag is set; otherwise the **test** flag is cleared and the destination is unchanged.

cm:unsigned-max-constant *destination source-constant length* [Operation]
 cm:unsigned-min-constant *destination source-constant length* [Operation]

The destination field is considered to be an unsigned integer. If the source-constant is greater (**cm:unsigned-max-constant**) or less (**cm:unsigned-min-constant**) than the destination, then the source-constant is copied to the destination and the **test** flag is set; otherwise the **test** flag is cleared and the destination is unchanged.

<code>cm:u= source1 source2 length</code>	[Operation]
<code>cm:u/= source1 source2 length</code>	[Operation]
<code>cm:u< source1 source2 length</code>	[Operation]
<code>cm:u<= source1 source2 length</code>	[Operation]
<code>cm:u> source1 source2 length</code>	[Operation]
<code>cm:u>= source1 source2 length</code>	[Operation]

The *source1* field is compared with the *source2* field, treating them as unsigned integers. The test flag is set if the *source1* field is equal to (`u=`), not equal to (`u/=`), less than (`u<`), less than or equal to (`u<=`), greater than (`u>`), or greater than or equal to (`u>=`) the *source2* field, and is cleared otherwise.

Of course, `u=` behaves identically to `=`, and `u/=` behaves identically to `/=`. They are provided primarily for reasons of symmetry.

Note that in COMMON LISP the not-equal operation may be written simply as `u/=` but when using Zetalisp on the Symbolics 3600 the notation `u//=` must be used because `/` is an escape character in Zetalisp.

<code>cm:u=constant source source-constant length</code>	[Operation]
<code>cm:u/=constant source source-constant length</code>	[Operation]
<code>cm:u<constant source source-constant length</code>	[Operation]
<code>cm:u<=constant source source-constant length</code>	[Operation]
<code>cm:u>constant source source-constant length</code>	[Operation]
<code>cm:u>=constant source source-constant length</code>	[Operation]

The *source* is compared with the *source-constant*, treating them as unsigned integers. The test flag is set if the *source* field is equal to (`=`), not equal to (`/=`), less than (`<`), less than or equal to (`<=`), greater than (`>`), or greater than or equal to (`>=`) the *source-constant* (a non-negative Lisp integer), and is cleared otherwise.

Of course, `u=constant` behaves identically to `=constant`, and `u/=constant` behaves identically to `/=constant`. They are provided primarily for reasons of symmetry.

Note that in COMMON LISP the not-equal operation may be written simply as `u/=constant` but when using Zetalisp on the Symbolics 3600 the notation `u//=constant` must be used because `/` is an escape character in Zetalisp.

<code>cm:unsigned-compare dest source1 source2 dlen slen1 slen2</code>	[Operation]
--	-------------

The two *source* fields are compared, and the value -1, 0, or 1 is stored in the destination according to whether the *source1* field is less than, equal to, or greater than the *source2* field, respectively. The length of the destination field must be at least 2.

The destination is always stored as a signed integer, despite the fact that both *source* fields are treated as unsigned integers.

The length of the *destination* field is specified by *dlen*; the length of the *source1* field is specified by *slen1*; the length of the *source2* field is specified by *slen2*.

<code>cm:unsigned-shift destination source dlen slen</code>	[Operation]
---	-------------

The contents of the *destination* field are shifted within that field by a distance determined

by the *source* field (which is a *signed* integer). In this way the *destination* field can be shifted by a different amount in each selected processor.

If the *source* field is positive, the shift is to the left (toward the most significant end of the field), and vacated low-order positions are cleared. The overflow flag is set if any 1-bit is shifted out, and otherwise is cleared.

If the *source* field is negative, the shift is to the right (toward the least significant end of the field), and vacated high-order positions are cleared. The overflow flag is always cleared in this case.

If the *source* field is zero, then the *destination* field is unaffected. The overflow flag is always cleared in this case.

7.4 Binary Operations on Floating-Point Numbers

Following the conventions of IEEE floating-point arithmetic, the overflow flag is “sticky” for floating-point operations; an operation may set the flag, but will never clear it. This is different from the behavior for integer arithmetic.

cm:f+ <i>destination source</i> &optional (<i>signif-len</i> 23) (<i>expt-len</i> 8)	[Operation]
cm:f- <i>destination source</i> &optional (<i>signif-len</i> 23) (<i>expt-len</i> 8)	[Operation]
cm:f* <i>destination source</i> &optional (<i>signif-len</i> 23) (<i>expt-len</i> 8)	[Operation]
cm:f/ <i>destination source</i> &optional (<i>signif-len</i> 23) (<i>expt-len</i> 8)	[Operation]

These are the standard four arithmetic operations on floating-point numbers.

In each case, the length *signif-len* must be not greater than the value of cm:*maximum-significand-length*, and the length *expt-len* must be not greater than the value of cm:*maximum-exponent-length*.

The result is rounded if necessary to make it fit into a significand of length *signif-len*. (If the mathematically correct result lies exactly between two representable floating-point numbers, then it is rounded to the floating-point value whose significand's least-significant bit is zero; this is the “round to even” rule.)

If the correct result cannot be represented because its exponent is too large (overflow) or too small (underflow), then the overflow flag is set and the contents of the *destination* field are unpredictable. Otherwise the overflow flag is unaffected. The test flag is unaffected by the operations cm:f+, cm:f-, and cm:f*; for the operation cm:f/, the test flag is set in the case of division by zero, and is unaffected otherwise.

Note that in COMMON LISP the division operation may be written simply as f/ but when using Zetalisp on the Symbolics 3600 the notation f// must be used because / is an escape character in Zetalisp.

cm:float-max <i>destination source</i>	[Operation]
&optional (<i>signif-len</i> 23) (<i>expt-len</i> 8)	
cm:float-min <i>destination source</i>	[Operation]
&optional (<i>signif-len</i> 23) (<i>expt-len</i> 8)	

If the source is greater (float-max) or less (float-min) than the destination, then the

source is copied to the destination and the test flag is set; otherwise the test flag is cleared and the destination is unchanged.

<code>cm:f= source1 source2 &optional (signif-len 23) (expt-len 8)</code>	[Operation]
<code>cm:f/= source1 source2 &optional (signif-len 23) (expt-len 8)</code>	[Operation]
<code>cm:f< source1 source2 &optional (signif-len 23) (expt-len 8)</code>	[Operation]
<code>cm:f<= source1 source2 &optional (signif-len 23) (expt-len 8)</code>	[Operation]
<code>cm:f> source1 source2 &optional (signif-len 23) (expt-len 8)</code>	[Operation]
<code>cm:f>= source1 source2 &optional (signif-len 23) (expt-len 8)</code>	[Operation]

The source is compared with the destination, treating them as floating-point numbers. The test flag is set if the *source1* field is equal to (`f=`), not equal to (`f/=`), less than (`f<`), less than or equal to (`f<=`), greater than (`f>`), or greater than or equal to (`f>=`) the *source2* field, and is cleared otherwise.

Note that in COMMON LISP the not-equal operation may be written simply as `f/=` but when using Zetalisp on the Symbolics 3600 the notation `f//=` must be used because `/` is an escape character in Zetalisp.

<code>cm:float-compare dest source1 source2 dlen</code>	[Operation]
<code>&optional (signif-len 23) (expt-len 8)</code>	

The two source fields are compared, and the value -1, 0, or 1 is stored in the destination according to whether the *source1* field is less than, equal to, or greater than the *source2* field, respectively. The length of the destination field must be at least 2.

The destination is always stored as a signed integer, despite the fact that both source fields are treated as floating-point numbers.

The length of the *destination* field is specified by *dlen*; the length of the *source1* field is specified by *slen1*; the length of the *source2* field is specified by *slen2*.

Chapter 8

Other Simple Operations

Each function in this chapter operates on one or more fields. The operation is performed in every processor selected by the context flag.

If a flag is not explicitly mentioned in the individual description of an operation, then the operation leaves the flag unaffected.

8.1 Movement of Fields

`cm:move destination source length` [Operation]
`cm:move-always destination source length` [Operation]

The *source* field is copied into the *destination* field. The function `cm:move` copies conditionally; `cm:move-always` copies unconditionally.

This operation is unusual in that it always operates correctly in the face of overlapping fields; it always behaves as if all the bits of the *source* were copied to an invisible buffer, and the buffer were then copied into the *destination* field.

These operations may be used to operate on the flags.

`cm:move-constant destination source-constant length` [Operation]
`cm:move-constant-always destination source-constant length` [Operation]

The low-order bits of the source-constant are copied into the *destination* field. The function `cm:move-constant` copies conditionally; `cm:move-constant-always` copies unconditionally.

An appropriate way to clear a field is to move the constant zero into it:

`(cm:move-constant destination 0 length)`

These operations may be used to operate on the flags. The standard way to turn on the context flag in all processors is

`(cm:move-constant-always cm:context-flag 1 1)`

`cm:float-move` *destination source* [Operation]
 &optional (*signif-len* 23) (*expt-len* 8)

The *source* field is copied into the *destination* field as a floating-point number. This operation is unusual in that it always operates correctly in the face of overlapping fields; it always behaves as if all the bits of the *source* were copied to an invisible buffer, and the buffer were then copied into the *destination* field.

This operation is no different from `cm:move` except that it allows specification of the length in terms of significand and exponent lengths. (When IEEE standard floating-point arithmetic is supported, `cm:float-move` will detect NAN values, but `cm:move` will not.)

`cm:float-move-constant` *destination source-constant* [Operation]
 &optional (*signif-len* 23) (*expt-len* 8)

The *source-constant* is copied into the *destination* field as a floating-point number. The *source-constant* may be any Lisp floating-point number. The *source-constant* need not be of the same format as that specified by *signif-len* and *expt-len*.

No indication is given if the value of the *source-constant* cannot be represented in the format specified by *signif-len* and *expt-len*. If the exponent can be represented adequately but the significand cannot, then precision will be lost but the value will be substantially correct (that is, rounding or truncation may occur).

`cm:float-move-decoded-constant` *destination signif-constant* [Operation]
 expt-constant sign-constant &optional (*signif-len* 23) (*expt-len* 8)

The operands *signif-constant*, *expt-constant*, and *sign-constant* are Lisp integers representing a constant floating-point value in the same manner as the values produced by the Common Lisp function `integer-decode-float`. This floating-point value is copied into the *destination* field as a floating-point number. The *signif-constant* need not be between $2^{\text{signif-len}-1}$ and $2^{\text{signif-len}} - 1$; it may be any non-negative integer.

No indication is given if the value of the floating-point constant cannot be represented in the format specified by *signif-len* and *expt-len*. If the exponent can be represented adequately but the significand cannot, then precision will be lost but the value will be substantially correct (that is, rounding or truncation may occur).

`cm:move-reversed` *destination source length* [Operation]

The *destination* field receives the reverse of the *source* field. The low-order bit of the *source* field is copied to the high-order bit of the *destination* field, etc. This works when the *source* and *destination* fields are the same field, but not for other cases of overlap.

`cm:pop-and-discard` *n* [Operation]

The non-negative integer value *n* is added to the stack pointer, thereby popping *n* bits from the stack and discarding them. This operation is unconditional.

cm:push-space *n* [Operation]

The non-negative integer value *n* is subtracted from the stack pointer, thereby pushing *n* bits onto the stack without initialization. Stack limit checking is performed. This operation is unconditional.

8.2 Arrays

The operations in this section allow each selected processor to treat a portion of its own memory as an array of elements. The element to be fetched or stored is determined by a per-processor index.

cm:aref *dest base index dest-length index-length array-length* [Operation]
element-length

This is a primitive form of array reference, for arrays stored in the memory of individual processors. Each processor has an array index stored in the field starting at *index*. This is used to index into an array which starts at *base*. The element indexed to, of length *dest-length*, is copied into *dest* in all selected processors. Thus different processors may access different elements of their arrays. More precisely, the field starting at *base + i × element-length*, where *i* is the unsigned number stored at *index*, is copied to a position starting at *dest*, in all selected processors.

The argument *array-length* is one greater than the largest allowed value of the index. Those processors which have index values greater than or equal to *array-length* will not alter the value of the destination field; they will also clear test flag. All processors in which the index field is less than *array-length* will set test flag. The argument *element-length* is the length of individual elements of the array. Usually this will be the same as *dest-length*, but for certain applications it is worthwhile for it to differ. For example, to extract a 4-character substring from a string of 8-bit characters, *dest-length* is 32 but *element-length* is 8.

This operation requires one bit of temporary storage. The size of the gap must be at least this large when the operation is performed, and the contents of the gap may be arbitrarily altered.

cm:aset *source base index source-length index-length array-length* [Operation]
element-length

This is a primitive form of array modification. Each processor has an array index stored in the field *index*. This is used to index into an array which starts at *base*. The field starting at *source* is copied into this element in all selected processors. More precisely, the field starting at *source*, of length *source-length*, is copied into the field starting at *base + i × element-length*, where *i* is the unsigned number stored at *index*, in all selected processors.

The argument *array-length* is one greater than the largest allowed value of the index. Those processors which have index values greater than or equal to *array-length* will not transfer any data; they will also clear test flag. All processors in which the index field

is less than *array-length* will set *test* flag. The argument *element-length* is the length of individual elements of the array.

This operation requires one bit of temporary storage. The size of the gap must be at least this large when the operation is performed, and the contents of the gap may be arbitrarily altered.

8.3 Miscellaneous Operations

`cm:unsigned-random destination length &optional limit` [Operation]

A random unsigned value is placed into the destination in each selected processor; each selected processor ideally receives an independently chosen random value. If no *limit* is specified, any of the 2^{length} possible values may be chosen; if a *limit* is specified, then each value will be less than that limit.

`cm:latch-leds source` [Operation]

`cm:latch-leds-always source` [Operation]

The specified 1-bit field is read from every selected processor (or every processor, for the *-always* version) and used to determine which LEDs should be illuminated. There is one LED associated with each group of 16 physical processors; each physical processor has some number of virtual processors. Two virtual processors belong to the same group if their virtual processor numbers agree in their twelve most significant bits. A LED is illuminated if every selected virtual processor in the group has a 0 in the selected *source* field (that is, the fields are combined for each group by a logical NOR operation).

These operations may be used to operate on the flags.

Note that the pattern will actually persist in the lights only if (`cm:set-system-leds-mode nil`) has been performed; otherwise the Connection Machine system software will present other patterns in the lights.

Chapter 9

Cooperative Computations

9.1 Global Operations

Each function in this section operates on one field in every selected processor, combining their values to produce a single result that is then reported back to the front-end computer and returned as the value of the operation; that is, each operation is a value-returning function and not just a procedure executed for its effect on the Connection Machine memory.

Returning a value to the front end presents the possibility of overflow. Consider first the case of integers. In the Lisp implementation, overflow cannot occur; the result is delivered as a “bignum” if necessary.

In the case of floating-point computation, overflow may occur in converting the result to a format acceptable to the front end. In the Lisp implementation, an extra value is returned to indicate whether or not overflow occurred.

9.1.1 Global Operations on Bit Fields

<code>cm:global-logand</code>	<i>source length</i>	[Operation]
<code>cm:global-logior</code>	<i>source length</i>	[Operation]
<code>cm:global-logand-always</code>	<i>source length</i>	[Operation]
<code>cm:global-logior-always</code>	<i>source length</i>	[Operation]

The specified field is examined in all selected processors (for `cm:global-logand` and `cm:global-logior`) or in all processors (for the `-always` versions of the functions). The fields are combined by performing bitwise logical AND (`global-logand`) or INCLUSIVE OR (`global-logior`) operations. The resulting combined field is then treated as an unsigned integer and returned to the front-end computer.

These operations may be used to operate on the flags.

If no processors are selected, then the identity for the operation is returned; this value is $2^{\text{length}} - 1$ for `global-logand` and 0 for `global-logior`.

`cm:global-count source` [Operation]
`cm:global-count-always source` [Operation]

The specified bit is examined in all selected processors (for the function `cm:global-count`) or all processors, regardless of selection (for the function `cm:global-count-always`). The number of bits that are 1 rather than 0 is returned as a Lisp integer. This is similar to using `cm:global-unsigned-add` on a one-bit field; however, `cm:global-count` may be used to operate on flags, whereas `cm:global-add` cannot.

If no processors are selected, then 0 is returned.

9.1.2 Global Operations on Signed Integers

`cm:global-add source length` [Operation]

The specified field is examined in all selected processors, and all the field values, treated as signed integers, are added together and returned as a Lisp integer.

In the Lisp implementation, overflow cannot occur; increased precision is used as necessary to guarantee an accurate result.

If no processors are selected, then 0 is returned.

This operation requires $length + 1$ bits of temporary storage. The size of the gap must be at least this large when the operation is performed, and the contents of the gap may be arbitrarily altered.

`cm:global-max source length` [Operation]
`cm:global-min source length` [Operation]

The specified field is examined in all selected processors. If at least one processor is selected, then the largest (`global-max`) or smallest (`global-min`) of the field values, treated as signed integers, is returned as a signed integer.

The test flag is set in every selected processor whose field is equal to the finally computed value, and is cleared in all other selected processors.

Overflow cannot occur; the result is delivered as a bignum if necessary.

If no processors are selected, then $-2^{length-1}$ is returned for `global-max`, and $2^{length-1} - 1$ is returned for `global-min`. In the Lisp implementation, a second value of `t` is returned to indicate that no processors were selected.

9.1.3 Global Operations on Unsigned Integers

`cm:global-unsigned-add source length` [Operation]

The specified field is examined in all selected processors, and all the field values, treated as unsigned integers, are added together and returned as an unsigned integer.

In the Lisp implementation, overflow cannot occur; increased precision is used as necessary to guarantee an accurate result.

If no processors are selected, then 0 is returned.

This operation requires $length + 1$ bits of temporary storage. The size of the gap must be at least this large when the operation is performed, and the contents of the gap may be arbitrarily altered.

cm:global-unsigned-max *source length* [Operation]

cm:global-unsigned-min *source length* [Operation]

The specified field is examined in all selected processors. If at least one processor is selected, then the largest (global-unsigned-max) or smallest (global-unsigned-min) of the field values, treated as unsigned integers, is returned as a Lisp integer.

The test flag is set in every selected processor whose field is equal to the finally computed value, and is cleared in all other selected processors.

Overflow cannot occur; the result is delivered as a bignum if necessary.

If no processors are selected, then 0 is returned for global-unsigned-max, and $2^{length} - 1$ is returned for global-unsigned-min. In the Lisp implementation, a second value of *t* is returned to indicate that no processors were selected.

9.1.4 Global Operations on Floating-Point Numbers

cm:global-float-max *source* &optional (*signif-len* 23) (*expt-len* 8) [Operation]

cm:global-float-min *source* &optional (*signif-len* 23) (*expt-len* 8) [Operation]

The specified field is examined in all selected processors. If at least one processor is selected, then the largest (global-float-max) or smallest (global-float-min) of the field values, treated as floating-point numbers, is returned as a floating-point number. The result coming from the Connection Machine system is lengthened or shortened as necessary to accommodate the front-end-machine floating-point format. In the Lisp implementation, a floating-point format that can contain the result without overflow or loss of accuracy is used if possible.

Overflow cannot occur during the main calculation, but it may occur in reducing the result to fit the format of the front-end machine. In the Lisp implementation three values are returned in this case; the second value is *nil*, and the third value is *t* to indicate that overflow occurred.

If no processors are selected, then a very large negative value is returned for global-max, and a very large positive value is returned for global-min. (When IEEE floating-point arithmetic is implemented, these values will be minus infinity and plus infinity, respectively.) In the Lisp implementation, a second value of *t* is returned to indicate that no processors were selected (and of course in this case overflow cannot occur).

9.2 Enumeration

cm:max-scan *destination source length* [Operation]

cm:unsigned-max-scan *destination source length* [Operation]

cm:float-max-scan *destination source length* [Operation]

Within each selected processor (call its virtual cube address *a*), the *destination* receives the maximum of the *source* fields (treated as signed integers, unsigned integers,

or floating-point numbers, respectively) of all selected processors whose virtual cube address is less than or equal to a . In other words, these are max-prefix operations over the set (ordered by cube address) of selected processors.

For example, suppose that processors 3, 4, 7, and 9 are selected and no others, and their *source* fields respectively contain the values 6, 2, 8, 5. A call to `cm:unsigned-max-scan` would produce the results 6, 6, 8, and 8 in the respective *destination* fields.

The operations `cm:max-scan` and `cm:unsigned-max-scan` each require 135 bits of temporary storage if *length* is greater than 64, and otherwise require no temporary storage. The operation `cm:float-max-scan` requires 136 bits of temporary storage if *length* is greater than 64, and otherwise requires one bit of temporary storage. The size of the gap must be at least this large when the operation is performed, and the contents of the gap may be arbitrarily altered.

`cm:plus-scan destination source dlen slen` [Operation]

`cm:unsigned-plus-scan destination source dlen slen` [Operation]

Within each selected processor (call its virtual cube address a), the *destination* receives the sum (signed or unsigned, respectively) of the *source* fields of all selected processors whose virtual cube address is less than or equal to a . In other words, these are sum-prefix operations over the set (ordered by cube address) of selected processors.

For example, suppose that processors 3, 4, 7, and 9 are selected and no others, and that each of these processors happens to have its own cube address in the *source* field. A call to `cm:unsigned-plus-scan` would produce the results 3, 7, 14, and 25, in the respective *destination* fields.

In order to avoid the question of overflow, the following arbitrary restriction is imposed: the length *dlen* of the destination must be at least as great as the sum of *slen* and the value of `cm:*cube-address-length*`.

These operations require $slen + dlen + 2c + 3$ bits of temporary storage, where c is the value of `cm:*cube-address-length*`. The size of the gap must be at least this large when the operation is performed, and the contents of the gap may be arbitrarily altered.

`cm:enumerate destination length` [Operation]

`cm:enumerate-and-count destination length` [Operation]

For each selected processor, a different unsigned integer is stored in the destination field. If n processors are selected, then each one gets a different integer between 0 and $n - 1$, inclusive. The integers are assigned according to the ordering of virtual cube addresses. By this is meant the following: if the processor with virtual cube address p_1 is assigned integer k_1 by the `cm:enumerate` operation, and the processor with virtual cube address p_2 is assigned k_2 , then $k_1 < k_2$ if and only if $p_1 < p_2$.

This implies that selecting the same set of processors will always produce the same enumeration, and a variety of other useful properties.

The overflow flag is set if the integer for a given processor cannot be represented as an unsigned integer of the specified *length*.

The operation `enumerate-and-count` not only performs the enumeration, but also returns to the front end, as an unsigned integer, the number of selected processors, as if

(cm:global-count cm:context-flag) had been performed. The operation `enumerate` simply performs the enumeration and returns no useful value.

9.3 Sorting

cm:rank	destination key destination-length key-length	[Operation]
cm:unsigned-rank	destination key destination-length key-length	[Operation]
cm:float-rank	destination key destination-length	[Operation]
	&optional (key-signif-len 23) (key-expt-len 8)	

This operation determines the ordering necessary to sort the key fields of the selected processors. It does not actually move the data so as to sort it, but merely indicates where the data should be moved so as to sort it.

In more detail: The *destination* field in each selected processor receives, as an unsigned integer, the rank of that processor's key within the set of all selected keys. The smallest key has rank 0, the next smallest has rank 1, and so on; the largest key has rank $n - 1$ where n is the number of selected processors. Thus the sequence of operations

```
(cm:rank rank key ranklen keylen)
(cm:send result rank key keylen)
```

would cause the data originally at location *key* within the selected processors to end up, in sorted order, at location *result* within processors 0 through $n - 1$; the `cm:rank` operation would determine the ranking, and the `cm:send` operation would use this ranking to reorder the data in memory. (An advantage of decoupling the rank determination from the reordering process is that the data to be moved may be much larger than the key that determines the ordering, and indeed it may be desirable to reorder the other data but not the key itself. In this way ranking and reordering each need operate only on the relevant data.)

The `cm:rank` operation treats the *key* as a signed integer; `cm:unsigned-rank` treats the *key* as an unsigned integer; and `cm:float-rank` treats the *key* as a floating-point number.

The *destination-length* must be at least as large as the base-two logarithm of the value of `cm:*user-number-of-processors-limit*`.

The *key-length* must be not greater than $m - c - 1$, where m is the value of `cm:*maximum-message-length*` and c is the value of `cm:*cube-address-length*`.

These operations each require $2c + \text{key-length} + 3$ bits of temporary storage, where c is the value of `cm:*cube-address-length*`. The size of the gap must be at least this large when the operation is performed, and the contents of the gap may be arbitrarily altered.

Chapter 10

Interprocessor Communication

10.1 General Communication through the Router

The communications functions in this section transmit data in a general fashion that effectively allows any processor to communicate directly with any other processor.

<code>cm:send-with-overwrite</code>	<i>destination cube-address source length</i>	[Operation]
	<code>&optional (time-limit nil)</code>	
<code>cm:send-with-logior</code>	<i>destination cube-address source length</i>	[Operation]
	<code>&optional (time-limit nil)</code>	
<code>cm:send-with-logand</code>	<i>destination cube-address source length</i>	[Operation]
	<code>&optional (time-limit nil)</code>	
<code>cm:send-with-logxor</code>	<i>destination cube-address source length</i>	[Operation]
	<code>&optional (time-limit nil)</code>	
<code>cm:send-with-add</code>	<i>destination cube-address source length</i>	[Operation]
	<code>&optional (time-limit nil)</code>	
<code>cm:send-with-max</code>	<i>destination cube-address source length</i>	[Operation]
	<code>&optional (time-limit nil)</code>	
<code>cm:send-with-min</code>	<i>destination cube-address source length</i>	[Operation]
	<code>&optional (time-limit nil)</code>	
<code>cm:send-with-unsigned-max</code>	<i>destination cube-address source length</i>	[Operation]
	<code>&optional (time-limit nil)</code>	
<code>cm:send-with-unsigned-min</code>	<i>destination cube-address source length</i>	[Operation]
	<code>&optional (time-limit nil)</code>	
<code>cm:send</code>	<i>destination cube-address source length</i>	[Operation]
	<code>&optional (time-limit nil)</code>	

For every selected processor p_s , a message *length* bits long is sent from that processor to the processor p_d whose absolute cube address is stored at location *cube-address* in the memory of processor p_s . The message is taken from the field starting at location *source* within processor p_s and is stored into the field at location *destination* within processor p_d .

As an example, one might write


```

(cm:push-space (+ cm:*cube-address-length* 32))
(cm:my-cube-address (cm:stack 32))
(cm:-constant (cm:stack 32) 1 cm:*cube-address-length*)
(cm:move (cm:stack 0) this-value 32)
(cm:* (cm:stack 0) that-value 32)
(cm:send-with-overwrite 43 (cm:stack 32) (cm:stack 0) 32)
(cm:pop-and-discard (+ cm:*cube-address-length* 32))

```

to cause every selected processor to send the product of the 32-bit quantities at addresses `this-value` and `that-value` to the 32-bit field at location 43 in the processor whose cube address is one less than that of the sender.

This operation is conditional, but whether a message is sent depends only on the context flag of the originating processor p_o ; the message, once transmitted to processor p_d , is stored regardless of the context flag of processor p_d .

The *length* must be not greater than the value of `cm:*maximum-message-length*`.

If more than one message is sent to a given processor, then the operations differ in their treatment of collisions.

The `send-with-overwrite` operation will store one of the messages sent and will discard all the others without notice.

The `send-with-logand` operation will combine all messages and the original contents of the destination field with a bitwise logical AND operation. To receive the logical AND of only the messages, the destination area should first be set to all-ones in all processors that might receive a message.

The `send-with-logior` operation is similar but combines incoming messages with a bitwise logical INCLUSIVE OR operation. To receive the logical INCLUSIVE OR of only the messages, the destination area should first be cleared in all processors that might receive a message.

The `send-with-logxor` operation is similar but combines incoming messages with a bitwise logical EXCLUSIVE OR operation. To receive the logical EXCLUSIVE OR of only the messages, the destination area should first be cleared in all processors that might receive a message.

The `send-with-add` operation is similar, but adds incoming messages together. The `overflow` flag is set in a destination processor if overflow occurs at any step. All other processors have the `overflow` flag cleared unconditionally. The `carry` flag is not affected. To receive the sum of only the messages, the destination area should first be cleared in all processors that might receive a message.

The `send-with-max`, `send-with-min`, `send-with-unsigned-max`, and `send-with-unsigned-min` operations are similar, but combine incoming messages using signed or unsigned max or min operations. The `test` flag is not affected by the max and min operations, but rather is set as described below. When doing a `send-with-max` or `send-with-unsigned-max`, to receive the maximum of only the messages, the destination area should first be set to the minimum possible number of the appropriate type and length in all processors that might receive a message. When doing a `send-with-min` or `send-with-unsigned-min`, to receive the minimum of only the messages, the destination area

should first be set to the maximum possible number of the appropriate type and length in all processors that might receive a message.

The `send` operation is similar, but combines incoming messages in an unpredictable manner. This operation may be used when the programmer can guarantee that no processor will receive more than one message. Using this operation when it is appropriate may speed message delivery. The destination area need not be prepared.

For any of these sending operations, every virtual processor that receives one or more messages will have its `test` flag unconditionally set to 1. Every processor that receives no messages will have its `test` flag cleared.

If the optional argument *time-limit* is supplied and is non-nil, the delivery operation will be terminated at a fixed time, rather than being allowed to run until completion. The value of *time-limit* determines the time that will be allowed for messages to be delivered. If the time limit is too short to deliver all the messages, some will not arrive at their destinations. It will be as though those messages were never sent. Which messages will be delivered is unpredictable in any given case, but certain properties are guaranteed. If *time-limit* is 0, no messages will be delivered. For some sufficiently large value of *time-limit*, all messages will be delivered; this value depends on the distribution of selected processors, and the contents of their *cube-address* fields. Two identical calls to a sending function, with identical patterns of selected processors and absolute cube addresses, will deliver exactly the same set of messages. For any distribution of selected processors and absolute cube addresses, increasing *time-limit* will not decrease the set of messages which are delivered. It is an error for the value of *time-limit* to be anything other than nil or a non-negative integer.

<code>cm:store-with-overwrite</code>	<i>memory-address source length</i>	[Operation]
	&optional (<i>time-limit nil</i>)	
<code>cm:store-with-logior</code>	<i>memory-address source length</i>	[Operation]
	&optional (<i>time-limit nil</i>)	
<code>cm:store-with-logand</code>	<i>memory-address source length</i>	[Operation]
	&optional (<i>time-limit nil</i>)	
<code>cm:store-with-logxor</code>	<i>memory-address source length</i>	[Operation]
	&optional (<i>time-limit nil</i>)	
<code>cm:store-with-add</code>	<i>memory-address source length</i>	[Operation]
	&optional (<i>time-limit nil</i>)	
<code>cm:store-with-max</code>	<i>memory-address source length</i>	[Operation]
	&optional (<i>time-limit nil</i>)	
<code>cm:store-with-min</code>	<i>memory-address source length</i>	[Operation]
	&optional (<i>time-limit nil</i>)	
<code>cm:store-with-unsigned-max</code>	<i>memory-address source length</i>	[Operation]
	&optional (<i>time-limit nil</i>)	
<code>cm:store-with-unsigned-min</code>	<i>memory-address source length</i>	[Operation]
	&optional (<i>time-limit nil</i>)	
<code>cm:store</code>	<i>memory-address source length</i> &optional (<i>time-limit nil</i>)	[Operation]

For every selected processor *p*, a message *length* bits long is sent to a destination

specified by the absolute virtual memory address stored at location *memory-address* in the memory of processor p_s . This virtual memory address specifies a virtual processor p_d and a bit address b_d within that processor. The virtual memory address field consists of the absolute cube address of the destination processor followed by the memory within that processor. The message is taken from the field starting at location *source* within processor p_s and is stored into the field at location b_d within processor p_d .

This operation is conditional, but whether a message is sent depends only on the context flag of the originating processor p_s ; the message, once transmitted to processor p_d , is stored regardless of the context flag of processor p_d .

The *length* must be not greater than the value of `cm:*maximum-message-length*`.

If more than one message is sent to a given processor, then the operations differ in their treatment of collisions. In this respect the `cm:store-...` operations are entirely analogous to the corresponding `cm:send-...` operations described above.

For any of these sending operations, every virtual processor that receives one or more messages will have its *test* flag unconditionally set to 1. Every processor that receives no messages will have its *test* flag cleared.

The optional argument *time-limit* behaves as described above, in the description of `cm:send`.

These operations require $c + 3$ bits of temporary storage, where c is the value of `cm:*cube-address-length*`. The size of the gap must be at least this large when the operation is performed, and the contents of the gap may be arbitrarily altered.

`cm:get destination cube-address source length` [Operation]

For every selected processor p_d , a message *length* bits long is sent to p_d from the processor p_s whose absolute cube address is stored at location *cube-address* in the memory of processor p_s . The message is taken from the field starting at location *source* within processor p_s and is stored into the field at location *destination* within processor p_d .

Note that more than one selected processor may request data from the same source processor p_s , in which case the same data is sent to each of the requesting processors.

The *length* must be not greater than the value of `cm:*maximum-message-length*`.

This operation is conditional, but whether a message is sent depends only on the context flag of the requesting processor p_d ; the message is transmitted regardless of the context flag of the source processor p_s .

Collisions cannot occur with this operation. The *test* flag is not affected.

This operation requires $2c + 3$ bits of temporary storage, where c is the value of `cm:*cube-address-length*`. The size of the gap must be at least this large when the operation is performed, and the contents of the gap may be arbitrarily altered.

`cm:fetch destination memory-address length` [Operation]

For every selected processor p_d , an absolute virtual memory address stored in p_d at bit address *memory-address* specifies a source virtual processor p_s and a bit address b_s within that processor. A message *length* bits long is sent to p_d from the processor p_s . The message is taken from the field starting at location b_s within processor p_s and is stored into the field at location *destination* within processor p_d .

Note that more than one selected processor may request data from the same source processor p_s , possibly from the same field and possibly from different fields.

The *length* must be not greater than the value of `cm:*maximum-message-length*`.

This operation is conditional, but whether a message is sent depends only on the context flag of the requesting processor p_d ; the message is transmitted regardless of the context flag of the source processor p_s .

Collisions cannot occur with this operation. The test flag is not affected.

This operation requires $2c + v + 3$ bits of temporary storage, where c is the value of `cm:*cube-address-length*` and v is the value of `cm:*virtual-memory-address-length*`. The size of the gap must be at least this large when the operation is performed, and the contents of the gap may be arbitrarily altered.

10.2 Communication through the NEWS Grid

The communications functions in this section transmit data through the two-dimensional Connection Machine NEWS grid. They are considerably more efficient, when applicable, than using the general router mechanism.

<code>cm:get-from-north</code> <i>destination source length</i>	[Operation]
<code>cm:get-from-east</code> <i>destination source length</i>	[Operation]
<code>cm:get-from-west</code> <i>destination source length</i>	[Operation]
<code>cm:get-from-south</code> <i>destination source length</i>	[Operation]

Into the *destination* field of a given selected virtual processor is stored a copy of the *source* field of the virtual processor to its north, south, east, or west, respectively, regardless of whether the source processor is selected.

These operations may be used to operate on the flags.

The *length* must be not greater than the value of `cm:*maximum-integer-length*`.

These operations each require *length* bits of temporary storage. The size of the gap must be at least this large when such an operation is performed, and the contents of the gap may be arbitrarily altered.

<code>cm:get-from-north-always</code> <i>destination source length</i>	[Operation]
<code>cm:get-from-east-always</code> <i>destination source length</i>	[Operation]
<code>cm:get-from-west-always</code> <i>destination source length</i>	[Operation]
<code>cm:get-from-south-always</code> <i>destination source length</i>	[Operation]

Into the *destination* field of a given virtual processor is stored a copy of the *source* field of the virtual processor to its north, south, east, or west, respectively, regardless of whether the source or destination processor is selected.

These operations may be used to operate on the flags.

The *length* must be not greater than the value of `cm:*maximum-integer-length*`.

10.3 Addresses and Address Transformations

cm:my-cube-address *destination* [Operation]

For each selected processor, the absolute virtual cube address of that processor is stored into the destination. The number of bits stored depends on the number of virtual processors in use; this number is made available to the user in the global variable **cm:*cube-address-length***, which is initialized by the operation **cm:cold-boot**.

cm:my-x-address *destination* [Operation]

cm:my-y-address *destination* [Operation]

For each selected processor, the virtual NEWS x-address or y-address of that processor is stored into the destination. The number of bits stored depends on the number of virtual processors in use; the numbers are made available to the user in the global variables **cm:*x-news-address-length***, and **cm:*y-news-address-length***, which are initialized by the operation **cm:cold-boot**.

cm:x-from-cube *destination source* [Operation]

cm:y-from-cube *destination source* [Operation]

For each selected processor, a virtual cube address is taken from the *source* field and the corresponding virtual x address or y address is extracted and stored into the destination field. The number of bits read and stored depends on the number of virtual processors in use; the numbers are made available to the user in the global variables **cm:*cube-address-length***, **cm:*x-news-address-length***, and **cm:*y-news-address-length***, which are initialized by the operation **cm:cold-boot**.

cm:front-end-x-from-cube *cube-address* [Function]

cm:front-end-y-from-cube *cube-address* [Function]

Each of these operations is performed entirely within the front-end computer. The argument, a non-negative integer, is taken to be a virtual cube address. The corresponding virtual x address or y address is extracted and returned.

cm:cube-from-x-y *destination x-source y-source* [Operation]

For each selected processor, virtual x and y addresses are taken from the *x-source* and *y-source* fields and the corresponding virtual cube address is constructed and stored into the destination field. The number of bits read and stored depends on the number of virtual processors in use; the numbers are made available to the user in the global variables **cm:*cube-address-length***, **cm:*x-news-address-length***, and **cm:*y-news-address-length***, which are initialized by the operation **cm:cold-boot**.

cm:front-end-cube-from-x-y *x-address y-address* [Function]

This operation is performed entirely within the front-end computer. The arguments, non-negative integers, are taken to be virtual x and y addresses within the virtual NEWS grid. The corresponding virtual cube address is constructed and returned.

cm:enumerate-for-rendezvous *destination* [Operation]

For each selected processor, a different virtual cube address is stored in the destination field. These cube addresses are heuristically chosen so as to try to optimize router performance in the event that a message is sent to each of the chosen addresses simultaneously. Cube addresses are chosen in a consistent manner. By this the following is meant. Suppose two different sets of processors A and B are enumerated for rendezvous. Let E_A be the set of cube addresses assigned to the processors in A , and similarly for E_B . Without loss of generality assume that $|A| \leq |B|$. Then it is guaranteed that $E_A \subseteq E_B$. The processors at those cube addresses can then be used as intermediaries for parallel communication.

This all amounts to saying that **cm:enumerate-for-rendezvous** is almost exactly like **cm:enumerate** (and indeed could correctly be implemented simply by using **cm:enumerate**), but does not guarantee to order processors according to their virtual cube addresses.

cm:processor-cons *destination source* [Operation]

For each selected processor, the virtual cube address of a distinct *free* processor is (perhaps) stored in the destination field.

A processor is considered to be *free* if its 1-bit *source* field contains a 1, regardless of whether or not it is selected. As much as possible, free processors are paired with selected processors, and for each pair three actions occur:

1. The selected processor receives the address of the free processor.
2. The **test** flag of the selected processor is set.
3. The *source* bit of the free processor is set to 0.

Any selected processor that is not paired with a free processor has its **test** flag cleared. The **test** flag of an unselected processor is not affected.

The *source* bit of any free processor that is not paired with a selected processor is not affected.

It is possible for a given processor to be both selected and free, in which case it might or might not be paired with itself. It is expected that for many applications selected processors will not be free.

This operation requires $8c + 13$ bits of temporary storage, where c is the value of **cm:cube-address-length**. The size of the gap must be at least this large when the operation is performed, and the contents of the gap may be arbitrarily altered.

Chapter 11

Memory Data Transfers

11.1 Transfer of Single Items

The operations described in this section transfer a single data item to or from a single field in a single specified processor.

<code>cm:read-from-processor</code>	<i>address source length</i>	[Operation]
<code>cm:unsigned-read-from-processor</code>	<i>address source length</i>	[Operation]
<code>cm:float-read-from-processor</code>	<i>address source</i>	[Operation]
	<i>&optional (signif-len 23) (expt-len 8)</i>	

A field of bits, starting at virtual memory address *source* and continuing for *length* bits, is read from the memory of the virtual processor whose virtual cube address is *address*. This operation is performed unconditionally, without regard for the context flag of the processor.

For `read-from-processor`, this field of bits is interpreted as a signed integer, and is returned as a Lisp integer. When the front-end computer is the Symbolics 3600, this operation is most efficient when the *address* is a multiple of 16 and the *length* is either 16 or 32.

For `unsigned-read-from-processor`, this field of bits is interpreted as an unsigned integer, and is returned as a Lisp integer. When the front-end computer is the Symbolics 3600, this operation is most efficient when the *address* is a multiple of 16 and the *length* is 16. (A *length* of 32 is equally efficient, provided that the most significant bit of the value read is a zero; otherwise a “bignum” must be produced.) This operation may be used to read the flags, provided that *length* is equal to 1.

For `float-read-from-processor`, this field of bits is interpreted as a floating-point number, and is returned as a Lisp floating-point number. Overflow may occur in reducing the number read to fit the format of the front-end computer. A second Lisp value is returned, which is `t` if overflow occurred during the reduction and `nil` if no overflow occurred. When the front-end computer is the Symbolics 3600, this operation is most efficient when the *address* is a multiple of 16, *signif-len* is 23, and *expt-len* is 8.


```

cm:write-to-processor address destination value length [Operation]
cm:unsigned-write-to-processor address destination value length [Operation]
cm:float-write-to-processor address destination value [Operation]
                        &optional (signif-len 23) (expt-len 8)

```

The specified *value* is stored into a field of bits, starting at virtual memory address *destination* and continuing for *length* bits, within the memory of the virtual processor whose virtual cube address is *address*. This operation is performed unconditionally, without regard for the context flag of the processor.

For *write-to-processor*, the value should be a Lisp integer; the low-order bits of the integer are stored. When the front-end computer is the Symbolics 3600, this operation is most efficient when the *address* is a multiple of 16 and the *length* is either 16 or 32.

The operation *unsigned-write-to-processor* behaves identically to *write-to-processor*, and is provided primarily for reasons of symmetry. This operation may be used to write the flags, provided that *length* is equal to 1.

For *float-write-to-processor*, the *value* should be a Lisp floating-point number. Overflow may occur in reducing the number read to fit the format of the specified field, but such overflow is not detected. When the front-end computer is the Symbolics 3600, this operation is most efficient when the *address* is a multiple of 16, *signif-len* is 23, and *expt-len* is 8.

11.2 Transfer of Arrays

The array transfer operations form a symmetrical group arranged according to three orthogonal categories:

1. *Data type*. This refers to the type of item transferred.

- (a) signed integers
- (b) unsigned integers
- (c) floating-point numbers

The case of integers being transferred to or from a packed Lisp integer array is likely to be implemented in a particularly efficient manner.

2. *Direction*.

- (a) Operations with *read* in their names transfer data from virtual processors in the Connection Machine system to an array in the front-end computer.
- (b) Operations with *write* in their names transfer data from an array in the front end to virtual processors in the Connection Machine system.

3. *Order*. Every array transfer transfers one item to or from each virtual processor within a set of virtual processors. This set may be designated in either of two ways:

- (a) A linear range of virtual cube addresses
- (b) A subrectangle of the virtual NEWS grid

cm:read-array-by-cube-addresses *array array-offset* [Operation]
 cube-address-start cube-address-end source length

cm:write-array-by-cube-addresses *array array-offset* [Operation]
 cube-address-start cube-address-end destination length

cm:read-array-by-news-addresses *array array-x-offset* [Operation]
 array-y-offset x-start y-start x-end y-end source length

cm:write-array-by-news-addresses *array array-x-offset* [Operation]
 array-y-offset x-start y-start x-end y-end destination length

Signed integer values are transferred between the specified *array* and Connection Machine processors. The array may be a general S-expression array containing signed integers, or may be a specialized integer-element array (such as the kind called **art-8b** on the Symbolics 3600).

For cube-oriented transfers, the *array* should be one-dimensional, and an index *j* ranges from 0 to *cube-address-end* - *cube-address-start* - 1, inclusive. The data from virtual processor *cube-address-start* + *j* is transferred to or from array element *array-offset* + *j*. Note that, following the conventions of Common Lisp, *cube-address-start* is an inclusive address (the address of the first processor involved in the transfer), but *cube-address-end* is an exclusive address (one more than the address of the last processor involved in the transfer). The total number of values transferred is therefore *cube-address-end* - *cube-address-start*.

For NEWS-oriented transfers, the *array* should be two-dimensional, and indices *j* and *k* range respectively from 0 to *x-end* - *x-start* - 1, inclusive, and from 0 to *y-end* - *y-start* - 1, inclusive. The data from the virtual processor whose NEWS coordinates are (*x-start* + *j*, *y-start* + *k*) is transferred to or from the array element whose indices are (*array-x-offset* + *j*, *array-y-offset* + *k*). Note that, following the conventions of Common Lisp, *x-start* and *y-start* are inclusive addresses, but *x-end* and *y-end* are exclusive addresses. The total number of values transferred is therefore

$$(x-end - x-start) \times (y-end - y-start)$$

Such a transfer copies a subrectangle of the virtual NEWS grid to or from a subrectangle of an array in the front end. This can be particularly useful for copying to or from Connection Machine memory an image to be displayed on a bit-mapped raster-scan graphics terminal.

The *source* or *destination* parameter specifies the memory address within each processor of the field to be transferred, considered as a signed integer. The *length* parameter specifies the length of the field.

For **write** operations, the **overflow** flag is set in any processor for which the transferred integer cannot be represented in the specified field, and is cleared in any processor for which the transferred integer can be represented. The **overflow** flag is unaffected in any processor not involved in the transfer.

In the Lisp environment, overflow cannot occur for read operations; bignums are used in the front-end machine if necessary.

These operations are unconditional, performed without regard to the context flag.

cm:unsigned-read-array-by-cube-addresses *array array-offset* [Operation]
cube-address-start cube-address-end source length

cm:unsigned-write-array-by-cube-addresses *array array-offset* [Operation]
cube-address-start cube-address-end destination length

cm:unsigned-read-array-by-news-addresses *array array-x-offset* [Operation]
array-y-offset x-start y-start x-end y-end source length

cm:unsigned-write-array-by-news-addresses *array array-x-offset* [Operation]
array-y-offset x-start y-start x-end y-end destination length

These operations are exactly like the versions described above for signed integers, but deal in unsigned integer values.

These operations are unconditional, performed without regard to the context flag.

cm:float-read-array-by-cube-addresses *array array-offset* [Operation]
cube-address-start cube-address-end source
&optional (signif-len 23) (expt-len 8)

cm:float-write-array-by-cube-addresses *array array-offset* [Operation]
cube-address-start cube-address-end destination
&optional (signif-len 23) (expt-len 8)

cm:float-read-array-by-news-addresses *array array-x-offset* [Operation]
array-y-offset x-start y-start x-end y-end source
&optional (signif-len 23) (expt-len 8)

cm:float-write-array-by-news-addresses *array array-x-offset* [Operation]
array-y-offset x-start y-start x-end y-end destination
&optional (signif-len 23) (expt-len 8)

These operations are exactly like the versions described above for signed integers, but deal in floating-point numbers. Instead of a single *length* parameter, the two parameters *signif-len* and *expt-len* specify the floating-point format, as usual.

These operations are unconditional, performed without regard to the context flag.

Overflow *can* occur for read operations, even in the Lisp environment; no indication of this condition is given.

Chapter 12

Housekeeping Operations

12.1 Stack Limit, Pointer, and Upper Bound

The operations for setting the stack limit l , pointer p , and upper bound u enforce the constraint that $0 \leq l \leq p \leq u \leq \text{cm:user-memory-address-limit*}$.

cm:get-stack-pointer [Operation]

The stack pointer, which is the memory address of the topmost (lowest-addressed) bit on the stack, is returned.

cm:set-stack-pointer *place* [Operation]

The stack pointer is set to *place*, which must be a memory address between the stack limit (inclusive) and the stack upper bound (inclusive).

cm:reset-stack-pointer [Operation]

The stack pointer is reset so as to empty the stack. It then is equal to the stack upper bound.

cm:get-stack-limit [Operation]

The stack limit, which is the memory address of the lowest bit of the gap, is returned.

cm:set-stack-limit *place* [Operation]

The stack limit is set to *place*, which must be a memory address between zero (inclusive) and the stack pointer (inclusive).

cm:get-stack-upper-bound [Operation]

The stack upper bound, which is the memory address of the lowest bit of the upper data area (that is, one bit higher than the highest-addressed bit of the stack area), is returned.

`cm:set-stack-upper-bound place` [Operation]

The stack upper bound is set to *place*, which must be a memory address between the stack pointer (inclusive) and the value of `cm:*user-memory-address-limit*` (inclusive).

12.2 Initializing the Random Number Generator

`cm:initialize-random-number-generator &optional seed` [Operation]

The generator of pseudo-random numbers used by the operation `cm:unsigned-random` is initialized. If a seed (a front-end integer) is provided, then that determines the initial state; if no seed is specified, then a value based on the date and time of day is used. Note that `cm:cold-boot` effectively calls `cm:initialize-random-number-generator` with no seed.

12.3 Controlling the Cabinet Lights (LEDs)

`cm:set-system-leds-mode mode` [Operation]

The lights on the front and back of the Connection Machine system cabinet can be controlled in a variety of ways. The `cm:set-system-leds-mode` operation selects what information will be displayed in the lights. If the specified *mode* is `nil`, then all the lights are turned off, and thereafter the user operations `cm:latch-leds` and `cm:latch-leds-always` may be used to control the lights. If the mode is not `nil`, then it should be one of the symbols in the list that is the value of `cm:*system-leds-modes*`, to select one of the system-supplied display modes. (The operations `cm:latch-leds` and `cm:latch-leds-always` may still be used when in a system-supplied display mode, but the user-specified pattern is unlikely to persist as it may be immediately altered by the system, depending on the mode.) The available modes may change over the course of time, but it will always work and produce some “interesting” display based on the contents of memory.

`cm:*system-leds-modes*` [Variable]

This is a list of all non-`nil` values that may validly be used as an argument to the `cm:set-system-leds-mode` operation.

12.4 Initializing the Connection Machine System

A single Connection Machine system can be shared among up to four front-end computers. At any given instance, the entire set of Connection Machine processors may be used by a single front-end computer, or the processors may be divided up so that each front end can use some fraction of the total number of processors.

Within a single Connection Machine system, the processors are divided up into portions of fixed size; each portion has either 8,192 or 16,384 processors. A single Connection Machine system can have up to four portions.

Associated with each Connection Machine portion is an independent microcontroller. The microcontroller is responsible for interpreting the macroinstructions issued by the front end. Associated with each front-end computer is a bus interface that contains two FIFO (first-in/first-out) queues; the IFIFO buffers macroinstructions and data issued from the front end to the microcontroller, and the OFIFO buffers data being returned to the front end for such operations as `cm:global-max` and `cm:read-array-by-cube-addresses`.

The front-end computer bus interfaces are connected to the microcontrollers through a 4×4 crossbar switch called the *Nexus*. (See figure 12.1.) Before a front-end computer can use a Connection Machine system, or part of it, it must logically attach itself to one or more microcontrollers by reconfiguring the Nexus. This of course must be done carefully so that ongoing computations by other front ends using other portions will not be disrupted. There are two macroinstructions, `cm:power-up` and `cm:detach`, that can disrupt users on other front ends, because they reset the state of the Nexus. Most other macroinstructions do not affect the Nexus directly; the few that do use a special negotiation protocol for the allocation and freeing of microcontrollers.

In all there are five operations for allocating and initializing Connection Machine processors for use by a front-end computer.

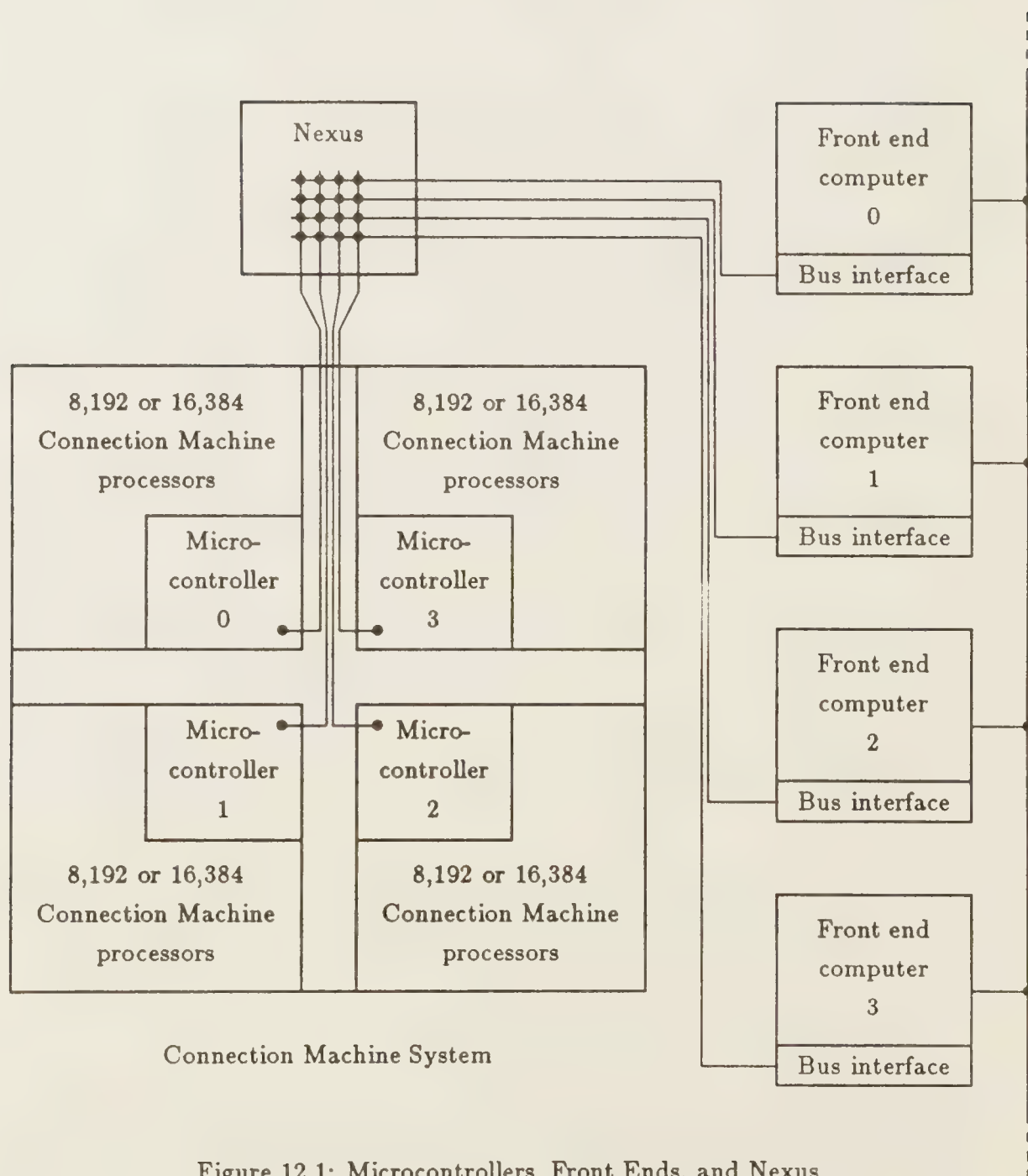
- `cm:power-up` is a once-only operation that resets the entire Connection Machine system, including the Nexus, and readies it for use by all of the front-end computers.
- `cm:attach` allocates hardware processors (and their associated microcontrollers) for use by a particular front end and configures them as virtual processors.
- `cm:cold-boot` completely resets the state of the hardware allocated to the executing front end, loads microcode, initializes system tables, and clears user memory.
- `cm:warm-boot` recovers from an error by resetting the dynamic state of the allocated hardware and possibly reloading microcode, but does not clear user memory.
- `cm:detach` releases any attached hardware processors so that they may be attached to other front-end computers (or perhaps reattached to the same front end in a different configuration). This operation may also be used to detach forcibly *another* front end from its Connection Machine processors.

The operations `cm:hardware-test-fast` and `cm:hardware-test-complete` are also provided, for the purpose of diagnostic testing.

`cm:power-up`

[Operation]

This function resets the state of the Nexus, causing all front-end computers to become logically detached from the Connection Machine system. When a Connection Machine system is first powered up or is to be completely reset for other reasons, this is the first operation to perform. Any of the front-end computers may be used to do it.



If users on other front-end computers are actively using the Connection Machine system, their computations will be disrupted. Normally all the front-end computers are connected not only through the Connection Machine Nexus but also through some sort of communications network; a front end that executes `cm:power-up` will attempt to send messages through this network to the other front-end computers on the same Nexus indicating that a `cm:power-up` operation is being performed.

`cm:attach` &optional *physical-size* [Operation]

This function is responsible for allocating Connection Machine processors for use by the front end. (To deallocate them, use `cm:detach`.)

If the *physical-size* argument is not specified, then the smallest possible amount of hardware will be allocated; this will be either 8,192 or 16,384 physical processors. Otherwise the *physical-size* argument must be one of the following:

:8k or 8192 Exactly 8,192 physical processors are to be allocated.

:16k or 16384 Exactly 16,384 physical processors are to be allocated.

:32k or 32768 Exactly 32,768 physical processors are to be allocated.

:64k or 65536 Exactly 65,536 physical processors are to be allocated.

:ucc0, :ucc1, :ucc2, or :ucc3 Exactly the specified microcontroller port is to be attached, regardless of whether that port controls 8,192 or 16,384 physical processors. (This option is useful primarily for hardware diagnostic procedures.)

:ucc0-1, :ucc2-3, or :ucc0-3 Exactly the specified microcontroller ports (0 and 1, 2 and 3, or all four) are to be attached, regardless of the number of physical processors involved. (This option is useful primarily for hardware diagnostic procedures.)

An error is signalled if the required number of physical processors or the required set of microcontroller ports is not available.

The value returned by `cm:attach` is the number of physical processors that were attached.

The variable `cm:*before-attach-initializations*` and the variable `cm:*after-attach-initializations*` contain sets of initializations that are respectively evaluated before and after anything else occurs.

`cm:cold-boot` &key :dimensions :microcode-version [Operation]

This operation completely resets the state of the hardware allocated to the executing front end, loads microcode, initializes system tables, and clears user memory. The `:microcode-version` argument specifies what set of microcode is to be loaded into the microcontroller(s). There are two choices for this argument: `:paris` (the default) specifies microcode that interprets the macroinstruction set, and `:diagnostics` specifies special microcode used for hardware maintenance.

The `:dimensions` argument must be an integer, a list of 1 or 2 integers, or unsupplied. (Passing `nil` as the value is the same as not supplying a value.) An integer or a list of one integer specifies the total number of *virtual* processors desired. A list of two integers specifies the desired size of the *virtual* NEWS grid. Each dimension must be a power of two.

If the `:dimensions` argument is unsupplied, then the configuration of virtual processors depends on the most recent `cm:cold-boot` or `cm:attach` operation preceding this one. If the most recent such operation was `cm:cold-boot`, then the same virtual processor configuration set up then will be used this time. If the most recent such operation was `cm:attach`, then the number of virtual processors will be equal to the number of physical processors, and the virtual NEWS grid will have the same shape as the physical NEWS grid.

Bootstrapping a Connection Machine system includes the following actions:

- Evaluating all initialization forms stored in the variable `cm:*before-cold-boot-initializations*`. This is done before anything else.
- Loading microcode into the Connection Machine microcontroller and initiating microcontroller execution.
- Clearing and initializing the memory of allocated Connection Machine processors.
- Initializing all of the global configuration variables described in section 3.6.
- Initializing the pseudo-random number generator by effectively invoking the operation `cm:initialize-random-number-generator` with no seed.
- Initializing the system lights-display mode by effectively invoking the operation `cm:set-system-leds-mode` with an argument of `t`.
- Evaluating all initialization forms stored in the variable `cm:*after-cold-boot-initializations*`. This is done after everything else.

If the cold-booting operation fails, then an error is signalled. If it succeeds, then three values are returned: the number of virtual processors, the number of physical processors, and the number of bits available for the user in each virtual processor. (These are exactly the values of the configuration variables `cm:*user-cube-address-limit*`, `cm:*physical-cube-address-limit*`, and `cm:*user-memory-address-limit*`.)

`cm:warm-boot`

[Operation]

This operation clears error status indicators for the attached Connection Machine hardware. It also clears the IFIFO and OFIFO in the bus interface and possibly loads fresh microcode into the attached microcontroller(s). The user memory areas in the Connection Machine system are not disturbed, but are checked for errors; any memory errors are reported. Certain system memory areas in the Connection Machine system are reinitialized, but the state of the pseudo-random number generator is not altered and

the system lights-display mode is not altered. The intent is to recover from an error condition while preserving as much of the machine state as possible.

This operation takes no arguments and returns no values. It signals an error if the warm-boot process was not successful.

There are two sets of initializations, kept in the variables `cm:*before-warm-boot-initializations*` and `cm:*after-warm-boot-initializations*`, that are evaluated before and after anything else occurs.

`cm:detach` &optional *front-end-name* [Operation]

Normally no *front-end-name* argument is specified. In this case the front end executing the call to `cm:detach` releases all Connection Machine hardware to which it had been attached, resetting relevant parts of the Nexus so that the front end can no longer issue macroinstructions to the Connection Machine system. (An error is signalled if in fact no hardware had been attached in the first place.) This use of `cm:detach` is the normal way of releasing attached hardware and will not disrupt users on other front ends.

If a *front-end-name* argument is specified, it must be the name of a front end that is connected to the same Connection Machine system (that is, Nexus) as the front end executing the call. Specifying the name of the front end that is executing the call has the same effect as specifying no argument; the front end is gracefully detached. But specifying the name of some other front end forcibly detaches that other front end, possibly disrupting any ongoing interaction with the Connection Machine system. The external communications network is used to send a message to the detached front end to inform its user that it has been forcibly detached.

There are two sets of initializations, kept in the variables `cm:*before-detach-initializations*` and `cm:*after-detach-initializations*`, that are evaluated before and after anything else occurs.

`cm:hardware-test-fast` [Operation]

This does a quick (less than two minutes) system verification of the attached hardware to assure that the macrocode set, microcode set, and hardware are in order. Warning: this test destroys all user memory state in the attached Connection Machine processors. This operation can be called no matter what microcode is currently loaded; when it finishes, the `:paris` microcode will be active.

The variable `cm:*before-hardware-test-fast-initializations*` and the variable `cm:*after-hardware-test-fast-initializations*` contain sets of initializations that are respectively evaluated before and after anything else occurs.

`cm:hardware-test-complete` [Operation]

This does a thorough system test of the attached hardware to assure that the macrocode set, microcode set, and hardware are in order. See the diagnostics manual and PARIS tests manual. Warning: this test destroys all user memory state in the attached Connection Machine processors. This operation can be called no matter what microcode is currently loaded; when it finishes, the `:paris` microcode will be active.

The variable `cm:*before-hardware-test-complete-initializations*` and the variable `cm:*after-hardware-test-complete-initializations*` contain sets of initializations that are respectively evaluated before and after anything else occurs.

<code>cm:*before-attach-initializations*</code>	[Variable]
<code>cm:*after-attach-initializations*</code>	[Variable]
<code>cm:*before-cold-boot-initializations*</code>	[Variable]
<code>cm:*after-cold-boot-initializations*</code>	[Variable]
<code>cm:*before-warm-boot-initializations*</code>	[Variable]
<code>cm:*after-warm-boot-initializations*</code>	[Variable]
<code>cm:*before-detach-initializations*</code>	[Variable]
<code>cm:*after-detach-initializations*</code>	[Variable]
<code>cm:*before-hardware-test-fast-initializations*</code>	[Variable]
<code>cm:*after-hardware-test-fast-initializations*</code>	[Variable]
<code>cm:*before-hardware-test-complete-initializations*</code>	[Variable]
<code>cm:*after-hardware-test-complete-initializations*</code>	[Variable]

These variables contains sets of initialization forms to be executed before or after the various hardware initialization operations described above.

`cm:add-initialization` *name-of-form* *form* *variable* [Operation]

This may be used to add an initialization form to a set of initializations. The *form* may be any executable Lisp form. The *name-of-form* should be a string. Adding two forms with the same name is permissible only if the forms are equal; otherwise an error is signalled. The *variable* should be the name of one of the initialization-set variables listed above, or it may be a list of such variable names, in which case the *form* is added to each of the sets.

Example: the *Lisp system performs the call

```
(cm:add-initialization
  "*COLD-BOOT Warnings"
  '(if (not *inside-*cold-boot-p*)
        (format t " %Warning: If you are using *Lisp, you must now call *COLD-BOOT.")
        (cm:*after-cold-boot-initializations* cm:*after-attach-initializations*)))
```

or one very much like it, where the variable `inside-*cold-boot-p` is managed in much the following manner:

```
(defvar *inside-*cold-boot-p* nil)
(defvar *first-*cold-boot* nil)

(defun *cold-boot (...)
  (let ((inside-*cold-boot-p t)) ;suppress initialization messages
    (when *first-*cold-boot*
      (cm:attach ...)
      (setq *first-*cold-boot* nil)))
```

```
(cm:cold-boot ...)
...)
```

cm:delete-initialization *name-of-form variable* [Operation]

This operation deletes the initialization form named by *name-of-form* from the initialization set (or sets) specified by *variable*. The arguments are specified in the same manner as the first and third arguments for **cm:add-initialization**.

12.5 Getting Information

cm:finger *&optional name (stream *standard-output*)* [Operation]

This prints a table to the specified *stream* indicating which front ends are using what portions of a given Connection Machine system.

If the name of a front end is specified, information is reported for only that front end. If the name of a Connection Machine system is specified, information is reported for all front ends connected to that Connection Machine system. If no name is specified (the argument is unsupplied or *nil*), then information is reported for all front ends connected to the same Connection Machine system as the executing front end.

Here is an example of what might be printed by **cm:finger**:

Connection Machine System Gemstone		Physical size: 64K processors with 4K RAM
Fred	Sapphire	Microcontroller Port (0 1) <-- 32768 physical processors
Wilma	Emerald	Not Attached to a Port
Betty	Ruby	Microcontroller Port (2) <-- 16384 physical processors
Barney	Topaz	Microcontroller Port (3) <-- 16384 physical processors

Here the Connection Machine system in question is named Gemstone, and it has 65,536 physical processors, each with 4,096 bits of memory. It is connected to four front ends named Sapphire, Emerald, Ruby, and Topaz, which are being used by users named Fred, Wilma, Betty, and Barney, respectively. Fred is using 32,786 physical processors; Betty and Barney are each using 16,384 processors; and Wilma is not using the Connection Machine system at all.

cm:time *form* [Macro]

The *form* is executed in the normal manner, but before the value is returned, timing information is printed out as for the Common Lisp **time** macro. In addition, timing information related to Connection Machine system performance is printed.

Index

cm:* operation 40
cm:*after-attach-initializations*
variable 77
cm:*after-cold-boot-initializations*
variable 77
cm:*after-detach-initializations*
variable 77
cm:*after-hardware-test-complete-
initializations* variable
77
cm:*after-hardware-test-fast-
initializations* variable
77
cm:*after-warm-boot-initializations*
variable 77
cm:*before-attach-initializations*
variable 77
cm:*before-cold-boot-
initializations* variable
77
cm:*before-detach-initializations*
variable 77
cm:*before-hardware-test-complete-
initializations* variable
77
cm:*before-hardware-test-fast-
initializations* variable
77
cm:*before-warm-boot-
initializations* variable
77
cm:*cube-address-length* variable 19
cm:*full-virtual-memory-address-
limit* variable
19
cm:*maximum-exponent-length* variable
20
cm:*maximum-integer-length* variable
19
cm:*maximum-message-length* variable
20
cm:*maximum-significand-length*
variable 20
cm:*physical-cube-address-length*
variable 19
cm:*physical-cube-address-limit*
variable 19
cm:*physical-x-dimension-limit*
variable 19
cm:*physical-x-news-address-length*
variable 19
cm:*physical-y-dimension-limit*
variable 19
cm:*physical-y-news-address-length*
variable 19
cm:*purely-virtual-cube-address-
length* variable
19
cm:*purely-virtual-x-news-address-
length* variable
19
cm:*purely-virtual-y-news-address-
length* variable
19
cm:*system-leds-modes* variable 71
cm:*user-cube-address-limit* variable
19
cm:*user-memory-address-length*
variable 19
cm:*user-memory-address-limit*
variable 19
cm:*user-x-dimension-limit* variable
19
cm:*user-y-dimension-limit* variable
19
cm:*virtual-memory-address-length*
variable 19
cm:*virtual-to-physical-processor-

- ratio* variable
19
- cm:*x-news-address-length* variable
19
- cm:*x-virtual-to-physical-processor-
ratio* variable
19
- cm:*y-news-address-length* variable
19
- cm:*y-virtual-to-physical-processor-
ratio* variable
19
- cm:+ operation 38
- cm:+carry operation 38
- cm:+constant operation 38
- cm:+flags operation 39
- cm:- operation 39
- cm:-borrow operation 39
- cm:-constant operation 39
- cm:/= operation 42
- cm:/=constant operation 42
- cm:< operation 42
- cm:<= operation 42
- cm:<=constant operation 42
- cm:<constant operation 42
- cm:= operation 42
- cm:=constant operation 42
- cm:> operation 42
- cm:>= operation 42
- cm:>=constant operation 42
- cm:>constant operation 42
- cm:abs operation 29
- cm:add operation 39
- cm:add-initialization operation 77
- cm:aref operation 52
- cm:aset operation 52
- cm:attach operation 74
- cm:ceiling operation 34
- cm:ceiling-and-remainder operation 41
- cm:ceiling-divide operation 40
- cm:cold-boot operation 74
- cm:compare operation 42
- cm:cube-from-x-y operation 64
- cm:delete-initialization operation 78
- cm:detach operation 76
- cm:enumerate operation 57
- cm:enumerate-and-count operation 57
- cm:enumerate-for-rendezvous operation
65
- cm:f* operation 48
- cm:f+ operation 48
- cm:f- operation 48
- cm:f/ operation 48
- cm:f/= operation 49
- cm:f< operation 49
- cm:f<= operation 49
- cm:f= operation 49
- cm:f> operation 49
- cm:f>= operation 49
- cm:fetch operation 62
- cm:finger operation 78
- cm:float operation 30
- cm:float-abs operation 33
- cm:float-compare operation 49
- cm:float-float-signum operation 33
- cm:float-max operation 48
- cm:float-max-scan operation 56
- cm:float-min operation 48
- cm:float-minusp operation 34
- cm:float-move operation 51
- cm:float-move-constant operation 51
- cm:float-move-decoded-constant
operation 51
- cm:float-negate operation 33
- cm:float-new-size operation 34
- cm:float-plusp operation 34
- cm:float-rank operation 58
- cm:float-read-array-by-cube-
addresses operation
69
- cm:float-read-array-by-news-
addresses operation
69
- cm:float-read-from-processor
operation 66
- cm:float-signum operation 33

- cm:float-sqrt operation 33
- cm:float-write-array-by-cube-addresses operation 69
- cm:float-write-array-by-news-addresses operation 69
- cm:float-write-to-processor operation 67
- cm:float-zerop operation 33
- cm:floor operation 34
- cm:floor-and-mod operation 41
- cm:floor-divide operation 40
- cm:front-end-cube-from-x-y function 64
- cm:front-end-gray-code-from-integer function 32
- cm:front-end-integer-from-gray-code function 33
- cm:front-end-x-from-cube function 64
- cm:front-end-y-from-cube function 64
- cm:get operation 62
- cm:get-from-east operation 63
- cm:get-from-east-always operation 63
- cm:get-from-north operation 63
- cm:get-from-north-always operation 63
- cm:get-from-south operation 63
- cm:get-from-south-always operation 63
- cm:get-from-west operation 63
- cm:get-from-west-always operation 63
- cm:get-stack-limit operation 70
- cm:get-stack-pointer operation 70
- cm:get-stack-upper-bound operation 70
- cm:global-add operation 55
- cm:global-count operation 55
- cm:global-count-always operation 55
- cm:global-float-max operation 56
- cm:global-float-min operation 56
- cm:global-logand operation 54
- cm:global-logand-always operation 54
- cm:global-logior operation 54
- cm:global-logior-always operation 54
- cm:global-max operation 55
- cm:global-min operation 55
- cm:global-unsigned-add operation 55
- cm:global-unsigned-max operation 56
- cm:global-unsigned-min operation 56
- cm:gray-code-from-integer operation 32
- cm:hardware-test-complete operation 76
- cm:hardware-test-fast operation 76
- cm:initialize-random-number-generator operation 71
- cm:integer-from-gray-code operation 33
- cm:integer-length operation 31
- cm:isqrt operation 30
- cm:latch-leds operation 53
- cm:latch-leds-always operation 53
- cm:logand operation 36
- cm:logand-always operation 36
- cm:logandc1 operation 37
- cm:logandc1-always operation 37
- cm:logandc2 operation 38
- cm:logandc2-always operation 38
- cm:logcount operation 31
- cm:logeqv operation 37
- cm:logeqv-always operation 37
- cm:logior operation 37
- cm:logior-always operation 37
- cm:lognand operation 37
- cm:lognand-always operation 37
- cm:lognor operation 37
- cm:lognor-always operation 37
- cm:lognot operation 29
- cm:lognot-always operation 29
- cm:logorc1 operation 38
- cm:logorc1-always operation 38
- cm:logorc2 operation 38
- cm:logorc2-always operation 38
- cm:logxor operation 37
- cm:logxor-always operation 37
- cm:max operation 41
- cm:max-constant operation 41
- cm:max-scan operation 56
- cm:min operation 41
- cm:min-constant operation 41
- cm:minusp operation 30
- cm:mod operation 41

- cm:move operation 50
- cm:move-always operation 50
- cm:move-constant operation 50
- cm:move-constant-always operation 50
- cm:move-reversed operation 51
- cm:multiply operation 40
- cm:my-cube-address operation 64
- cm:my-x-address operation 64
- cm:my-y-address operation 64
- cm:negate operation 30
- cm:new-size operation 30
- cm:plus-scan operation 57
- cm:plusp operation 30
- cm:pop-and-discard operation 51
- cm:power-up operation 72
- cm:processor-cons operation 65
- cm:push-space operation 52
- cm:rank operation 58
- cm:read-array-by-cube-addresses operation 68
- cm:read-array-by-news-addresses operation 68
- cm:read-from-processor operation 66
- cm:rem operation 41
- cm:reset-stack-pointer operation 70
- cm:round operation 34
- cm:round-and-remainder operation 41
- cm:round-divide operation 40
- cm:send operation 59
- cm:send-with-add operation 59
- cm:send-with-logand operation 59
- cm:send-with-logior operation 59
- cm:send-with-logxor operation 59
- cm:send-with-max operation 59
- cm:send-with-min operation 59
- cm:send-with-overwrite operation 59
- cm:send-with-unsigned-max operation 59
- cm:send-with-unsigned-min operation 59
- cm:set-stack-limit operation 70
- cm:set-stack-pointer operation 70
- cm:set-stack-upper-bound operation 71
- cm:set-system-leds-mode operation 71
- cm:shift operation 42
- cm:signum operation 30
- cm:store operation 61
- cm:store-with-add operation 61
- cm:store-with-logand operation 61
- cm:store-with-logior operation 61
- cm:store-with-logxor operation 61
- cm:store-with-max operation 61
- cm:store-with-min operation 61
- cm:store-with-overwrite operation 61
- cm:store-with-unsigned-max operation 61
- cm:store-with-unsigned-min operation 61
- cm:subtract operation 39
- cm:time macro 78
- cm:truncate operation 34
- cm:truncate-and-rem operation 41
- cm:truncate-divide operation 40
- cm:u* operation 44
- cm:u+ operation 43
- cm:u+carry operation 43
- cm:u+constant operation 43
- cm:u+flags operation 43
- cm:u- operation 44
- cm:u-borrow operation 44
- cm:u-constant operation 44
- cm:u/= operation 47
- cm:u/=constant operation 47
- cm:u< operation 47
- cm:u<= operation 47
- cm:u<=constant operation 47
- cm:u<constant operation 47
- cm:u= operation 47
- cm:u=constant operation 47
- cm:u> operation 47
- cm:u>= operation 47
- cm:u>=constant operation 47
- cm:u>constant operation 47
- cm:unsigned-add operation 44
- cm:unsigned-ceiling operation 34
- cm:unsigned-ceiling-and-remainder operation 46
- cm:unsigned-ceiling-divide operation 45
- cm:unsigned-compare operation 47

- cm:unsigned-float operation 32
- cm:unsigned-floor operation 34
- cm:unsigned-floor-and-mod operation 46
- cm:unsigned-floor-divide operation 45
- cm:unsigned-integer-length operation 32
- cm:unsigned-isqrt operation 31
- cm:unsigned-logcount operation 32
- cm:unsigned-max operation 46
- cm:unsigned-max-constant operation 46
- cm:unsigned-max-scan operation 56
- cm:unsigned-min operation 46
- cm:unsigned-min-constant operation 46
- cm:unsigned-mod operation 45
- cm:unsigned-multiply operation 45
- cm:unsigned-negate operation 31
- cm:unsigned-new-size operation 32
- cm:unsigned-plus-scan operation 57
- cm:unsigned-plusp operation 31
- cm:unsigned-random operation 53
- cm:unsigned-rank operation 58
- cm:unsigned-read-array-by-cube-addresses operation 69
- cm:unsigned-read-array-by-news-addresses operation 69
- cm:unsigned-read-from-processor operation 66
- cm:unsigned-rem operation 45
- cm:unsigned-round operation 34
- cm:unsigned-round-and-remainder operation 46
- cm:unsigned-round-divide operation 45
- cm:unsigned-shift operation 47
- cm:unsigned-subtract operation 44
- cm:unsigned-truncate operation 34
- cm:unsigned-truncate-and-rem operation 46
- cm:unsigned-truncate-divide operation 45
- cm:unsigned-write-array-by-cube-addresses operation 69
- cm:unsigned-write-array-by-news-addresses operation 69
- cm:unsigned-write-to-processor operation 67
- cm:unsigned-zerop operation 31
- cm:warm-boot operation 75
- cm:write-array-by-cube-addresses operation 68
- cm:write-array-by-news-addresses operation 68
- cm:write-to-processor operation 67
- cm:x-from-cube operation 64
- cm:y-from-cube operation 64
- cm:zerop operation 30

User Contributed Software

Users are welcome to try out
and comment upon this soft-
ware. The existence of this
document does not imply a
commitment by Thinking Ma-
chines to supporting user con-
tributed software.

PI, The Processor Inspector

by Jim Davis

September, 1986

The Processor Inspector is a window program for inspecting *LISP processors. It displays the values of a set of pvars and expressions in a set of processors. You can alter values of pvars with the mouse and keyboard. The PI frame includes a grid of cells, each of which displays values from a single processor. Every cell of the grid displays the same set of values. The bottom line of the cell displays the address of the cell, in either grid or cube coordinates. Every other line displays either a pvar or an expression. You change the value of a pvar in a processor by pointing at its value with the mouse, clicking Left, and typing in the new value.

Choosing What to Display

Since there are many cells on the frame, there are only a few lines in each cell, so there isn't room to display the entire state of the processor. You tell PI what to display with the menu item `Cell values`. When you click on this item, a special menu appears which has one line reserved for every line in the cell. By editing this menu, you specify what will be displayed in each cell.

When the menu is first exposed, you see three column headings ("Label", "Object", and "Format"), and two items labeled `new pvar` and `new expression`. When you click on `new pvar`, you get a menu of all defined pvars. If you select one, it's added to the next free line of the display. If you click on `new expression`, a window appears where you can type in any expression you want. This expression must yield a pvar, not a Lisp value.

After you've selected one of these, a line will be added for the choice you've made. This line shows the label, object, and format of the item you've chosen. If the object has a long name, it will be abbreviated. Both the label and format are formed automatically, but you can change either by clicking on it. If you shorten a label you'll have more room for the display of the value.

To delete an item, click Middle on it.

You can add as many items as there are lines on the grid. At present, there is room for six items. After you add as many as will fit, the `new pvar` and `new expression` items disappear, because you can't add any more.

Using the Map

In the upper right corner of the frame is the Map pane, which displays a view of the entire CM as a grid, showing every processor that meets a specified

predicate. You can display two predicate expression at once, one in black and one in gray. The predicate can be any *LISP expression that yields a boolean value. You set the predicate with the menu item `Set Map Expression`. You click `Left` to set the black expression, `Middle` to set the gray expression. For example, to show all the processors where the pvar `a` is between 3 and 10, you would click on the menu item and then type `(>=!! (!! 10) a (!! 3))`. At present, the maximum size of the map is 256 by 256, so PI will not work if you have more than this many processors.

If you are working with fewer than 256 x 256 processors, you may want to increase the scale on the map to make it easier for you to see individual processors. If you click `Left` on `Adjust Map Scale`, the map scale will expand to the largest scale that shows the entire CM. For example, if you're using an 8x4 grid, each processor will be represented as a block 32 pixels square. You can also zoom in on a region by clicking `Middle`. With this option you can position a rectangle anywhere on the map you like.

Choosing Processors to Display

You can refer to processors by either cube addresses or grid addresses. You choose which by clicking on either `Grid` or `Cube`. One or the other of these items will be highlighted in black to remind you which you've chosen.

Since there are far more processors in the CM than there are cells in PI's frame, you have to tell PI which processors you want. One way to do this is to click on `Show Processor`, which prompts you for an address using the addressing mode you've chosen. Another way is to point at a processor on the Map, and click on it.

By default, you select each processor to display individually. If you are using Grid addressing, you can also select an entire rectangular block of processors at once, by choosing the location of the upper left corner. This is known as `Block selection`. The default is `Individual selection`. To use block selection, click on `Block`. In this mode, the map indicates the current position by drawing a rectangle. This rectangle represents the processors that are visible in the cells of the grid. When you use individual selection, there is no requirement that displayed processors be adjacent on the grid. The rectangle is scaled to show the size of the grid relative to the entire map. On a 256 by 256 map it is scarcely visible.

Controlling the Format of Pvar Display

The format of display for a pvar depends upon what it holds. Field pvars are displayed as unsigned integers, boolean pvars as `T` or `NIL`, and general pvars as whatever they hold.

PI makes special provision for displaying field pvars that contain character data. You can tell PI to display a pvar as a character either by editing the format in the layout menu, or by placing a value on the `*lisp:pi-format` property of the symbol for the pvar.

```
(proclaim '(type (field-pvar 8) letter))
(*defvar letter (!! 0))
(setf (get 'letter 'pi-format) :character)
```

Automatic Updates

You can get PI to update its display periodically. To do this, click Right on Update, and type in a time in seconds.

How to Get PI

To get the PI window, type `select- π` (that's symbol-shift-P). You can also use `inspect-processors`, described below. Note that PI takes a minute to create its initial window. Be patient.

PI is automatically loaded when you load `*LISP`. The version that is automatically loaded may not be the most recent version. To get the experimental version, do `(make-system 'pi :noconfirm :silent)`. Do *not* use `ms`. If you get warnings about function definitions, type `P` to proceed.

Functions for Inspecting Processors

The Processor Inspector package also provides a few auxilliary functions which may prove helpful in debugging. They are all in package `PI`.

`inspect-processors` **&key** (*addressing* :cube) *position* (*block* nil) *pvars*

Exposes a Processor Inspector window, with specified *addressing* (:cube or :grid), at a specified *position*, with a specified set of *pvars*. The interpretation of *position* depends upon the value of *addressing*. It can be either a single address (a cube address, or a list of two grid coordinates) or a list of valid addresses. If *block* is `T`, then *addressing* must be :grid, and *position* must be a single address, since the very meaning of *block* is to display a rectangle of processors starting at a single grid address.

`display-expression` *form* **&optional** (*window* standard-output)
&key (*x-offset* 0) (*y-offset* 0) (*scale* 1)

Evaluates *form* in all processors, displays the result in *window* in the same manner the map does. If supplied, *x-offset* and *y-offset* specify the grid address

of the processor displayed in the upper left corner of the window, and *scale* is a magnification factor.

```
show-expression form &key (scale 1) (x-offset 0) (y-offset 0)  
                      (width (*lisp:dimension-size 0))  
                      (height (*lisp:dimension-size 1))
```

Pops up a temporary map window, displaying *form* as `display-expression` does. The window remains exposed until you either click upon it or until you select the window that it covers.

Known Bugs

- Changing a pvar doesn't change the Map display
- The Map fails if any dimension exceeds 256.

Release Notes

Thinking Machines Corporation

Release Notes

Connection Machine Software Release 11

November 1986

© 1986 Thinking Machines Corporation
All Rights Reserved

This notice is intended as a precaution against inadvertant publication and does not constitute an admission or acknowledgement that publication has occurred, nor constitute a waiver of confidentiality. The information and concepts described in this document are the proprietary and confidential property of Thinking Machines Corporation.

© 1986 Thinking Machines Corporation.

“Connection Machine” is a registered trademark of Thinking Machines Corporation.

“*LISP” and “PARIS” are trademarks of Thinking Machines Corporation.

The information in this document is subject to change without notice and should not be construed as a commitment by Thinking Machines Corporation. Thinking Machines Corporation reserves the right to make changes to any products described herein to improve functioning or design. Although the information in this document has been reviewed and is believed to be reliable, Thinking Machines Corporation does not assume responsibility or liability for any errors that may appear in this document. Thinking Machines Corporation does not assume any liability arising from the application or use of any information or product described herein.

Release Notes

Connection Machine Software Release 11

November 1986

How (Not) To Read Release Notes

With the latest Connection Machine Software release, Release 11, several changes have been made to improve *LISP in performance and functionality. This set of release notes, whose contents is briefly described below, is meant to be used in conjunction with *The Essential *LISP Manual*, to reference all *LISP information currently available.

The pages that really must be read are this one and the Table of Contents. The prudent user will then be able to quickly find the information needed at the moment.

- **New *LISP Functions**
- **New *LISP Function Arguments**

These two sections are especially worth reading. These new function descriptions do not appear elsewhere.

- ***LISP Hardware Version Changes**

This section describes a new argument of interest that is available only in the hardware version of *LISP. It contains other release notes as well.

- ***LISP Simulator Changes**

This short section notes a few differences between the simulator and the hardware versions of the new functions, as well as a few bug fixes to the simulator.

- **A Release Incompatibility**

This section is very important to anyone with a need to use Release 10 software on a Release 11 machine; those using only Release 11 may ignore it.

Notation Conventions

The notation conventions used for describing new functions follows the conventions used in *COMMON LISP: The Language* by Guy L. Steele, Jr., with one exception. The default values for keyword arguments are shown enclosed in italicized parentheses with the keyword. These parentheses are not typed when actually using the keywords; they appear in the description syntax just to show the default values. If a keyword is not specified, the default value is used; alternatively the keyword may be specified followed by a different valid value. Several examples in the text illustrate this.

Table of Contents

New *LISP Functions	1
Scan Functions	1
Variations in the *LISP Simulator	4
Block Data Transfer Between Arrays and Pvars	4
Variations in the *LISP Simulator	5
Initialization Lists For *cold-boot and *warm-boot	6
Other New Functions	6
 New *LISP Function Arguments	8
Global Variables For pretty-print-pvar	8
 *LISP Hardware Version Changes	9
Improvements in Performance	9
Multiple Processor Reads	9
Improvements in Functionality	10
Bug Fixes	10
Improvements in Error Messages	11
Obsolete Items	11
An Experimental Feature—Defining Structures	11
 *LISP Simulator Changes	14
Bug Fixes	14
 A Release Incompatibility	15

New *LISP Functions

Scan Functions

Two new functions are available, `scan!!` and `scan-grid!!`.

```
scan!! pvar function &key (:direction :forward) [Function]
      :segment-pvar segment-pvar
      (:include-self T)
```

For each selected processor, the value returned to that individual processor is the result of reducing the pvar values in all the processors preceding it. Its own pvar value is by default included in the reduction as well. “Reducing” in this context refers to the COMMON LISP function, `reduce`, which accepts two arguments, *function* and *sequence*. The `reduce` function applies *function*, which must be a binary associative function, to all the elements of the *sequence*. For example, if `+` were the *function* all the elements in *sequence* would be summed. In the case of a `scan!!` function the sequence becomes the pvar values contained in the ordered set of selected processors.

The `scan!!` argument *function* is one of the following associative binary *LISP functions: `+`!!, `and`!!, `or`!!, `max`!!, `min`!!, and `copy`!!. (Arbitrary binary functions which accept two pvar arguments and return a pvar result will be accepted as *function* arguments in a future release.) In the following illustration, `*` is any of these binary functions:

(self-address!!)	processor-selected?	value of pvar	result of scan
0	no	a	
1	yes	b	b
2	yes	c	b*c
3	no	d	
4	yes	e	(b*c)*e
5	no	f	
6	yes	g	((b*c)*e)*g
7	no	h	

If `*` were the function `+`!!, this would be a summation over the set of selected processors, ordered by cube address:

```
(self-address!!)           => 0 1 2 3 4 5 6 7 ...
(scan!! (self-address!!) '+!!)) => 0 1 3 6 10 15 21 28 ...
```

Normally, the value returned for the last selected processor is the result of applying the *function* to all the preceding selected processors and the last one. One may however, break up the processors into *segments*. A segment consists of

a sequence of processors in ascending cube-address order. A new segment of processors begins at each processor in which *segment-pvar* is non-NIL. Even if all segment pvars are NIL, however, there is always at least one segment beginning with the selected processor having the lowest cube address. The first processor in a segment always receives the value of *pvar* instead of the reduction of all the preceding processors. For example:

```
(self-address!!)          =>  0  1  2  3  4  5  6  7...
segment-pvar              => nil nil nil t  t  nil nil t
(scan!! (self-address!!) '+!!
:segment-pvar segment-pvar) =>  0  1  3  3  4  9  15 7...
```

In this example there are four segments. The first is 0, 1, 2; second is 3; third is 4, 5, 6; and fourth is 7... .

Unlike the other functions which can be used as arguments, *copy!!* exists only as a scanning *function*. It is used only in conjunction with *segment-pvar*. It will cause the value of *pvar* in the first processor of a segment to be copied into all the other processors of that segment. For example:

```
(self-address!!)          =>  0  1  2  3  4  5  6  7...
segment-pvar              => nil nil nil t  t  nil nil t
(scan!! (self-address!!) 'copy!!
:segment-pvar segment-pvar) =>  0  0  0  3  4  4  4  7...
```

The direction of the scanning is normally from lowest to highest cube-address. If the *direction* argument is *:backward*, then the scan is from highest to lowest cube-address. When scanning backwards, segments are sequences of processors in descending cube-address order. In this example, the segments consist of first, ...7, 6, 5; next, 4; and lastly, 3, 2, 1, 0.

```
(self-address!!)          =>  0  1  2  3  4  5  6  7...
segment-pvar              => nil nil nil t  t  nil nil t
(scan!! (self-address!!) '+!!
:segment-pvar segment-pvar
:direction :backward)     =>  6  6  5  3  4  18 13 7...
```

Normally, each processor receives the result of applying *function* to all the processors before it and including itself. The *include-self* keyword controls whether the value of a processor is included. When *:include-self* is NIL, there are two effects:

1. The value of each processor is the result of applying *function* to all processors before it, excluding itself.
2. The value of processors in which *segment-pvar* is non-NIL is the result of applying *function* to all the processors of the *previous* segment.

Following are two examples:

(self-address!!)	=>	0	1	2	3	4	5	6	7...
segment-pvar	=>	nil	nil	nil	t	t	nil	nil	t
(scan!! (self-address!!) '+!!									
:segment-pvar segment-pvar									
:include-self t)	=>	0	1	3	3	4	9	15	7...
(scan!! (self-address!!) '+!!									
:segment-pvar segment-pvar									
:include-self nil)	=>	*	0	1	3	3	4	9	15...

In this first example, the case where `:include-self` is `T` is identical to the first `scan!!` example. The processor of cube-address 3 receives its own address added to no others, being the first processor in a new segment. In the second case, where `:include-self` is `NIL`, the value of processor 0 is undefined since there are no processors preceding it. Processor 3 receives a value that is the sum of the previous segment's processor addresses, $0 + 1 + 2$. Likewise, processor 7 receives a value that is the sum of the previous segment's processor addresses, $4 + 5 + 6$.

The second example illustrates the double effect achieved when `:include-self` is `NIL`, using the `max!!` function:

pvar	=>	1	10	5	20	3	4	5	6
segment-pvar	=>	nil	nil	nil	t	t	nil	nil	t
(scan!! pvar 'max!!									
:segment-pvar segment-pvar									
:include-self t)	=>	1	10	10	20	3	4	5	6
(scan!! pvar 'max!!									
:segment-pvar segment-pvar									
:include-self nil)	=>	*	1	10	10	20	3	4	5

Scanning can be accomplished using grid addressing as well as cube addressing.

```
scan-grid!! pvar function &key (:dimension :x) [Function]
              (:direction :forward)
              :segment-pvar segment-pvar
              (:include-self T)
```

The function `scan-grid!!` is similar to `scan!!` except that processors are scanned in grid order instead of cube order. The keyword argument *dimension* controls whether the scanning is done across rows or columns. It may be one of the symbols `:x` (rows) or `:y` (columns), or it may be a non-negative integer less than `*number-of-dimensions*`. A `:dimension` value of 0 corresponds to rows and a value of 1 corresponds to columns. Each row or column is a separate segment.

Variations in the *LISP Simulator

The *LISP Simulator does not yet support the *include-self* keyword argument in the scan functions.

Block Data Transfer Between Arrays and Pvars

Transferring data between the front-end computer and the Connection Machine may be done much more efficiently when the either source or destination of the transfer is an array. Instead of repetitively calling `pref`, or `setf` on `pref`, portions of the array can be moved in block mode using the functions described below.

```
pvar-to-array source-pvar &optional dest-array [Function]
  &key (:array-offset 0)
        (:cube-address-start 0)
        (:cube-address-end *number-of-processors-
                           limit*)
```

This function moves data from *source-pvar* into *dest-array* in cube-address order. If provided, *dest-array* must be one-dimensional. If a *dest-array* is not provided, an array is created of size *cube-address-end* minus *cube-address-start*. The data from *source-pvar* in processors *cube-address-start* through $1 - \text{cube-address-end}$ are written into *dest-array* elements starting with element *array-offset*. The result returned by `pvar-to-array` is *dest-array*.

```
array-to-pvar source-array &optional dest-pvar [Function]
  &key (:array-offset 0)
        (:cube-address-start 0)
        (:cube-address-end *number-of-processors-
                           limit*)
```

This function moves data from *source-array* to *dest-pvar*. The *source-array* must be one-dimensional. The other arguments behave the same way as in `pvar-into-array`. If a *dest-pvar* is not provided, `array-to-pvar` creates a destination pvar, in which case the call to the function must be within a function that accepts pvar expressions, such as `*set`. If a destination pvar is created, its value in processors to which `array-to-pvar` did not write is undefined. The value returned by this function is *dest-pvar*.


```

pvar-to-array-grid source-pvar &optional dest-array [Function]
  &key (:array-offset
        (make-list *number-of-dimensions*
                    :initial-element 0))
    (:grid-start
      (make-list *number-of-dimensions*
                  :initial-element 0))
    (:grid-end *current-cm-configuration*)

```

This function moves data from *source-pvar* into *dest-array* in grid address order. If provided, *dest-array* must have the same number of dimensions as the current Connection Machine configuration. If *dest-array* is not specified, an array is created with dimensions *grid-end* minus *grid-start*, where the subtraction is done component-wise to produce a list suitable for *make-array*. The data from *source-pvar* in the sub-grid defined by *grid-start* and *grid-end* as the upper and lower corners, respectively, are written into a similar sub-grid of *dest-array* starting with element *array-offset* as the upper corner. The arguments *array-offset*, *grid-start*, and *grid-end* must be lists of length **number-of-dimensions**. The value returned by *pvars-into-array-grid* is *dest-array*.

```

array-to-pvar-grid source-array &optional dest-pvar [Function]
  &key (:array-offset
        (make-list *number-of-dimensions*
                    :initial-element 0))
    (:grid-start
      (make-list *number-of-dimensions*
                  :initial-element 0))
    (:grid-end *current-cm-configuration*))

```

This function moves data from *source-array* to *dest-pvar* in grid address order. The number of dimensions *source-array* has must be equal to **number-of-dimensions**. The other arguments to this function behave the same way as in *pvar-to-array-grid*. If *dest-pvar* is *NIL*, *array-to-pvar-grid* creates a destination pvar, in which case the call to the function must be within a function that accepts pvar expressions, such as **set*. If a destination pvar is created, its value in processors to which *array-to-pvar-grid* did not write is undefined. The value returned is *dest-pvar*.

*Variations in the *LISP Simulator*

The **LISP* simulator does not support the functions *pvar-to-array-grid* and *array-to-pvar-grid* correctly.

Initialization Lists For **cold-boot* and **warm-boot*

Users can define a set of forms to be executed automatically before and after each execution of **cold-boot* and **warm-boot*. These user-defined initialization lists are stored in one or more of these variables:

before-*cold-boot-initializations [Variable]

after-*cold-boot-initializations [Variable]

before-*warm-boot-initializations [Variable]

after-*warm-boot-initializations [Variable]

New forms are added using the function *add-initialization*, and removed using *delete-initialization*.

add-initialization name-of-form form variable [Function]

The argument *name-of-form* gives a name to the form being added, and is a character string. The argument *form* may be any executable LISP form. Adding two forms with the same name is permissible only if the forms are the same according to the function *equal*; otherwise an error is signaled. The *variable* should be one of the initialization-list variables above, or it may be a list of such variables, in which case the *form* is added to each initialization list named.

delete-initialization name-of-form variable [Function]

This operation deletes the form named by *name-of-form* from the initialization list (or lists) specified by *variable*. The arguments are specified in the same manner as the first and third arguments for *add-initialization*.

Other New Functions

oddp!! integer-pvar [Function]

This predicate is true if the argument *integer-pvar* is odd (not divisible by two), and otherwise is false. It is an error if the contents of *integer-pvar* is not an integer in some processor.

evenp!! integer-pvar [Function]

This predicate is true if the argument *integer-pvar* is even (divisible by two), and otherwise is false. It is an error if the contents of *integer-pvar* is not an integer in some processor.

`signum!! number-pvar` [Function]

This function returns a pvar containing -1, 0, or 1 according to whether the number is negative, zero, or positive. For a floating-point number, the result will be a floating-point number of the same format.

`float!! number-pvar &optional other-pvar` [Function]

This function converts any number to a floating-point number. In processors in which *number-pvar* already contains floating-point numbers, those numbers are returned; otherwise, single-float numbers are produced. When the optional argument *other-pvar* is given, which must contain floating-point numbers, *number-pvar* is converted to the same format as *other-pvar*.

`rot!! integer-pvar n-pvar word-size-pvar` [Function]

This function returns *integer-pvar* rotated left $|n-pvar|$ bits. The rotation considers *integer-pvar* as a number of length *word-size-pvar* bits. This function is especially fast when *n-pvar* and *word-size-pvar* are both constant pvars.

`plusp!! number` [Function]

This predicate returns `T` if the argument *number* is greater than zero, and `NIL` otherwise. The argument must be non-complex.

`minusp!! number` [Function]

This predicate returns `T` if *number* is less than zero, and `NIL` otherwise. However, `(minusp!! (!! -0.0))` is always false. The argument must be non-complex.

`sin!! radians` [Function]

`cos!! radians` [Function]

The function `sin!!` returns the sine of the argument; `cos!!` returns the cosine. The argument is in radians, and may be complex. The current versions of these functions are preliminary. Much faster versions will be implemented in a future release.

`log!! number &optional base` [Function]

This function returns the logarithm of the argument *number* in the base *base*. If *base* is absent, the natural logarithm is returned.

New *LISP Function Arguments

Global Variables For pretty-print-pvar

Certain global variables affecting how the `pretty-print-pvar` function works were added.

```
pretty-print-pvar pvar [Function]
  &key (:mode *ppp-default-mode*)
        (:format *ppp-default-format*)
        (:per-line *ppp-default-per-line*)
        (:start *ppp-default-start*)
        (:end *ppp-default-end*)
```

`*ppp-default-mode*` [Variable]

This variable provides the default value for the keyword argument `:mode`. Its initial value is the keyword `:cube`. Its other legal value is `:grid`.

`*ppp-default-format*` [Variable]

This variable provides the default value for the keyword argument `:format`. Its initial value is the string `"-S "`.

`*ppp-default-per-line*` [Variable]

This variable provides the default value for the keyword argument `:per-line`. Its initial value is `NIL`.

`*ppp-default-start*` [Variable]

This variable provides the default value for the keyword argument `:start`. Its initial value is zero.

`*ppp-default-end*` [Variable]

This variable provides the default value for the keyword argument `:end`. Its initial value is `*number-of-processors-limit*`, and it is reset to this value whenever a `*cold-boot` is executed.

*LISP Hardware Version Changes

Improvements in Performance

- The functions `*or`, `*and`, and `cube-from-grid-address!!` are faster.
- The function `do-for-selected-processors` is more efficient.

Multiple Processor Reads

Several functions now take an additional optional argument, called *collision-mode*, that determines how cases are handled for efficiency where more than one processor is reading from a single processor. The order and type of the additional arguments (that is, whether they are optional or keywords) is subject to change in the next release.

`pref!! pvar-expression cube-address-pvar &optional collision-mode` [Function]

`pref-grid!! pvar-expression &rest grid-address-pvars` [Function]
`&optional collision-mode`
`&key :border-pvar border-pvar`

The `pref!!` functions, with the exception of `pref-grid-relative!!`, take the additional optional argument, *collision-mode*. The values allowed are `:collisions-allowed` (the default), `:no-collisions`, and `:many-collisions`. This argument allows *LISP to optimize calls to `pref!!` in the cases where each address is unique, as in `:no-collisions`, or when many addresses are identical, as in `:many-collisions`.

`:collisions-allowed`

This is the default mode. Each processor may access any other processor and multiple reads are allowed. The time required to complete this operation is proportional to the maximum number of processors reading from a single processor.

`:no-collisions`

This tells *LISP that no two processors will ever be caused to read from the same processor. It allows the Connection Machine to execute the read significantly faster than the `:collisions-allowed` case does. However, if two processors do attempt to read from the same processor, the behavior is unpredictable. Use with caution!

`:many-collisions`

This is useful when there are many processors reading from a single proces-

sor. *LISP uses a different algorithm to resolve the collisions. The result is that the `pref!!` almost takes constant time regardless of the routing pattern. That time is approximately the same as a delivery cycle using `:collisions-allowed` where thirty processors are reading from a single processor, although this may vary for different virtual processor ratios.

`*pset combiner value-pvar dest-pvar cube-address-pvar` [Function]
&optional `notify-pvar collision-mode`

`*pset-grid combiner value-pvar dest-pvar x-pvar y-pvar` [Function]
&optional `notify-pvar collision-mode`

The `*pset` functions, with the exception of `*pset-grid-relative`, also take an optional `collision-mode` argument. In this case it may take on only the two values `:collisions-allowed` (the default) or `:many-collisions`, and `:no-collisions` is specified as a `combiner` argument. As with `pref!!`, the time required to complete a delivery cycle is proportional to the maximum number of messages arriving at a single processor. When that number is greater than about twenty, it is better to specify `:many-collisions`. The cut-over point depends on the type of combiner and is discussed in *Supplement to the Essential *Lisp Manual* (writing in progress). The `combiner` argument `:no-collisions` causes messages to be delivered to processors somewhat faster than the other combiners.

A new optional argument to `*pset` and `*pset-grid` is `notify-pvar`. This argument must be a pvar; its value when `*pset` has finished executing is `T` in all processors into which a value is written, even if the value written happens to be the same as the pvar's current value, and is not affected in other processors.

Note that the syntax of `*pset-grid` will have to change in a future release because the assumption that there are always exactly two dimensions for grid addressing will be incorrect.

Improvements in Functionality

Bug Fixes

- It is now possible to execute a `*cold-boot` for any virtual processor ratio currently provided (see *Connection Machine Parallel Instruction Set (PARIS)* for information on virtual processor ratios). This has been tested for up to 8 x 8 virtual processors.
- The functions `*logior` and `*logand` didn't handle negative quantities properly in previous releases; they now do.
- The function `*sum` did not handle floating point numbers before; it has been fixed and now does.

- The function `allocate!!` now has slightly better error checking.
- All known bugs have been fixed in the functions `pvar-to-array` and `pvar-to-array-grid`.
- Recompiling a `*defvar` now gives a warning only once instead of twice.

Improvements in Error Messages

- Error reporting now prints the number of processors that have the incorrect type of value. In addition to providing `proceed` options to display the processors that are in error, the number of such processors is displayed.
- The error messages given by `oddp!!`, `evenp!!`, and `signum!!` have been fixed. The function `ash!!` now gives a more informative internal error message than “you lost.”
- The function `*funcall` now gives a reasonable error message when the function being called is invalid; it used to give an unwieldy message.

Obsolete Items

- The `pvar self-address!!` no longer exists because there is a function by the same name.
- The variable `*2/pi*` has been removed.

An Experimental Feature—Defining Structures

At present, the ability to create data types with named record structures and named components is an experimental feature, but it is explained here for the adventurous user. It is very likely this will change, and has been included in this release to encourage a discussion on processor objects and structures. Please mail any comments to `*LISP-USERS` on your local machine.

```
*defstruct (structure-name                                     [Macro]
           &key :conc-name (:immediate-data-type T))
           &rest (element-name initial-value :type pvar-type)
```

This function defines a structure. The keyword argument `:conc-name` behaves the same as COMMON LISP’s `defstruct` argument of the same name. The argument `:immediate-data-type` currently does nothing, but it must be included with a value of `T` when using `*defstruct`. It may mean something in the future.

Structure objects are created and returned in all selected processors by the following function, which is created automatically when a structure is defined:

make-structure-name!!

[Function]

This function provides keyword arguments for initializing the elements of the structure differently than they are defined in **defstruct*.

Structures are assigned to pvars using **set*:

**set pvar (make-structure-name!!)*

Given a pvar that contains structures, one may access the individual elements through a set of functions, one for each element, which are created automatically:

element-name!!

[Function]

References to the values of structure elements in specific processors are made by another set of functions which are created automatically:

pref-structure-name-element-name pvar address

[Function]

The function *setf* is used with *pref* above to modify a structure element.

structurep!! pvar

[Function]

This predicate checks to see if *pvar* contains any structures; *structurep!!* returns a pvar containing *T* in each processor in which *pvar* was a structure, and *NIL* otherwise.

The example on the following page defines two structures, an astronaut with age and sex data, and a ship with mass, x-location, and y-location.

- (1) The pvar `fleet` in odd cube-addressed processors is made to contain the astronaut structure.
- (2) In even cube-addressed processors `fleet` is made to contain the ship structure.
- (3) The value of the astronaut age in the first processor containing an astronaut is given to `astronaut-age-in-proc-1`.
- (4) The age in this structure is set to 99.
- (5) The ship mass and x-location in the first processor containing a ship are returned.
- (6) The ship mass in all processors containing the ship structure is set to 22.
- (7) The last line illustrates the structure predicate.

```

(*defstruct (astronaut :immediate-data-type t)
  (age 30 :type (field-pvar 7))
  (sex t :type boolean-pvar))

(*defstruct (ship :immediate-data-type t)
  (mass 0 :type (field-pvar 5))
  (x-loc -4 :type (signed-pvar 10))
  (y-loc -2 :type (signed-pvar 10)))

(*defvar fleet)

(1)  (*when (oddp!! (self-address!!)) (*set fleet (make-astronaut!!)))
(2)  (*when (evenp!! (self-address!!)) (*set fleet (make-ship!!)))
(3)  (setq astronaut-age-in-proc-1 (pref-astronaut-age fleet 1))
(4)  (setf (pref-astronaut-age fleet 1) 99)
(5)  (pref-ship-mass fleet 0) (pref-ship-x-loc fleet 0)
(6)  (*when (evenp!! (self-address!!))
      (setf (ship-mass!! fleet) (!! 22)))
(7)  (if (*or (structurep!! fleet))
        (format t "There are some structures in fleet"))

```

*LISP Simulator Changes

The *LISP Simulator does provide the new scan functions, but it does not support the *include-self* keyword argument. While it does have the new `pvar-to-array` and `array-to-pvar` functions, it does not support the grid-address versions, `pvar-to-array-grid` and `array-to-pvar-grid`, correctly.

Bug Fixes

- All known bugs in the macro expansion of the `and!!` and `or!!` macros were fixed.
- A bug preventing grid addressing to work in more than two dimensions was fixed.
- The code for the function `setf` and for the generation of trivial functions has been completely reworked. These changes are internal and should not cause any visible difference.
- Some code within the constructs `#+` and `#-` has been added for use in porting to SUN COMMON LISP.

A Release Incompatibility

This information is very important in the event that a previous software release must be used on a Release 11 Connection Machine system.

The main difference in hardware between Release 10 and Release 11 is in the error system. In an effort to isolate the chips that signal memory and instruction parity errors, modifications to the microcontroller and matrix boards have been made to allow the matrix boards to latch the error condition. The Release 11 software has been changed so that if such an error occurs, it scans the boards looking for the latched error. If it can find the board, it will include that information at the end of the error report. If it cannot find the error, it may suggest looking at the error lights. There is one new light for each matrix board, located in the vertical center and set back from the other lights.

With these hardware modifications, only Release 11 software should be used. If you need to use an earlier software release, you must turn off the error system by turning off (switch position down) the switch on the microcontroller. When using Release 11, that switch on the microcontroller should always be turned on (position up). Earlier releases will work with the error system turned off, except for one diagnostic in CM:Hardware-Test-Complete and CM:Hardware-Test-Fast which tests the error system. This switch is a small metal switch located on the microcontroller, about three inches up from the bottom of the microcontroller.

Summary of Correct Error Switch Positions

	<u>Hardware Release 10</u>	<u>Hardware Release 11</u>
Software Release 10	↑	↓
Software Release 11	↓	↑

Release Notes

Connection Machine Software Release 4.0A Field Test

April 1987

D R A F T

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, April 1987

The information in this document is subject to change without notice and should not be construed as a commitment by Thinking Machines Corporation. Thinking Machines Corporation reserves the right to make changes to any products described herein to improve functioning or design. Although the information in this document has been reviewed and is believed to be reliable, Thinking Machines Corporation does not assume responsibility or liability for any errors that may appear in this document. Thinking Machines Corporation does not assume any liability arising from the application or use of any information or product described herein.

PARIS, *LISP, and Connection Machine
are trademarks of Thinking Machines Corporation.
VAX and ULTRIX are trademarks of Digital Equipment Corporation.
UNIX is a trademark of AT&T Bell Laboratories.

Copyright © 1987 by Thinking Machines Corporation. All rights reserved.

Thinking Machines Corporation
245 First Street
Cambridge, MA 02142-1214
(617) 876-1111

New Release Numbering Scheme

Customer release numbering has been changed, as of Release 4.0, to be more compatible with standard naming conventions. Release 4.0, which implements the change, immediately follows Release 11. Hereafter release numbers will be in consecutive order. The four is the customer major release number, the major change in this case being the introduction of the VAX ULTRIX front end, and the zero is the customer minor release number, applicable when improvements and corrections to current software are released. The next customer release after 4.0 will be 4.1.

Components of Release 4.0A Field Test

Release 4.0 Field Test consists of a Connection Machine with both a Digital Equipment Corporation VAX and a Symbolics LISP machine as front ends. The front ends run the following software releases:

VAX	ULTRIX 2.0 operating system Connection Machine Software 4.0A Field Test
LISP machine	Symbolics Release 6.1 Connection Machine Software Release 11

The VAX comes with Lucid LISP. A user on the VAX has the option of entering the LISP environment and using the Connection Machine from there. It has the same functionality and user interface as the LISP environment on the LISP machine front end. However, the LISP environment on the VAX has Connection Machine Software Release 4.0A Field Test, while the LISP environment on the LISP machine is Connection Machine Software Release 11. Release 4.0A Field Test is a superset of Release 11; therefore, the Release 11 Notes included in the documentation set apply to both front ends' software releases. The last section in these release notes, "Changes from Release 11 to Release 4.0A," apply only to the VAX front end's LISP environment.

Known Restrictions

Restrictions to the LISP Environment on the VAX

- The LISP software on the VAX front end must be Lucid LISP; it cannot be from any other vendor.
- The Connection Machine software for the VAX front end is entirely COMMON LISP compatible; that is, all the system software written in LISP uses COMMON LISP as opposed to other LISP dialects.
- ULTRIX has a text segment limit of six megabyte. In Lucid LISP, some saved LISP images, or *worlds*, approach this limit. To address this problem, four different worlds are provided; their use depends on what you are doing. For example, to use *LISP on the Connection Machine, you would type `/usr/local/starlisp`, which would place you in a LISP world containing the *LISP software. To use PARIS or run diagnostics, you would type `/usr/local/lisp-paris`, which would place you in a LISP world containing PARIS and the diagnostics software. (The available worlds are described in *Connection Machine User's Guide: Using a UNIX System Front End* (Volume II).) This is different from the LISP world on a LISP machine, where all the software is available in a single saved image. The six-megabyte text segment limit may directly affect your ability to save working images. If you create programs on top of a world, it may not be possible to do an incremental disk save because of the text segment limit. Instead of creating and saving an image containing both the LISP world and your programs, you may have to load the LISP world of your choice and then load your own programs separately each time. If you do try to do a disk save, do *not* use the option shown:

```
(system:disk-save "filename" :full-gc t)
```

This option places more into the text segment and may cause the limit to be exceeded.

- Connection Machine Software Release 4.0A Field Test on the VAX does not support the full functionality of LISP available on the LISP machine. Specifically:

—Array transfer by NEWS addresses is not yet possible, which means the functions listed below cannot be used in LISP/PARIS (C/PARIS is not affected):

```
cm:float-read-array-by-news-addresses
cm:float-write-array-by-news-addresses
cm:read-array-by-news-addresses
cm:write-array-by-news-addresses
cm:unsigned-read-array-by-news-addresses
cm:unsigned-write-array-by-news-addresses
```


- Double-precision floating-point numbers may not be used. This is a limitation of Lucid LISP; it does not apply to C programming.
- The flavors implementation provided by Lucid is not included in the LISP worlds provided with this software release.

Restrictions to PARIS

The C Implementation

- When programming in C, numbers longer than 32 bits are not supported.
- There is no C/PARIS simulator on the VAX front end that can be used by C/PARIS programs. There is a LISP/PARIS simulator in the LISP environment, just as there is on the LISP machine front end.

The C and LISP Implementations

- The function `cm:float-sqrt` does not work properly when the fields overlap. This problem will be corrected in a future release.

Other Restrictions

- The VAX and the LISP machine do not communicate over the Ethernet connecting them, except for file transfer. As a result, a `cmfinger`—or `(cm:finger)`—command issued from either machine will determine whether the other front end is attached to the Connection Machine, but will not print the user name or the number of processors being used.
- The diagnostic test `(cm:hardware-test-fast)` can be run only from within a LISP environment, and it can be run on either the VAX or the LISP machine front end. Running this test takes about five minutes on the VAX and about thirty seconds on the LISP machine. The test `(cm:hardware-test-complete)` can only be run on the LISP machine and takes three to three and one half hours. (See *Connection Machine User's Guide* for the front end you wish to use to find out how to run diagnostics.)
- Virtual processor ratios from 1×1 up to 8×8 are allowed on either front end; either dimension may be one of 1, 2, 4, or 8. However, you should be aware that each physical processor has 4K bits of memory, and that the PARIS instructions requiring larger amounts of stack space might not have enough memory for virtual processor ratios of 4×4 (16 virtual processors per physical processor) and higher.

Documentation Update

In **LISP Release Notes: Connection Machine Software Release 11*, the **LISP* function `scan!!` is explained in terms of the COMMON LISP `reduce` function.

However, unlike `reduce`, the order in which the binary operation is performed on its operands is not guaranteed. The result is that for scans of floating-point pvars having values in different processors of widely differing orders of magnitude, precision may be lost. Scans performed on floating-point pvars having values in different processors of similar orders of magnitude are not affected.

Changes from Release 11 to Release 4.0A

The quick system diagnostic test, `(cm:hardware-test-fast)`, now prints an appropriate error message if the Connection Machine is not physically connected to the front end or if it is powered off. Previously, if a cable was accidentally pulled out of the Connection Machine, the system test did not detect it.

A meaningful error message is now printed in the event of a failure to cold boot the Connection Machine after attaching it.

PARIS Improvements

The PARIS simulator now supports integers greater than 63 bits in length.

A few PARIS functions, `cm:front-end-cube-from-x-y` for example, are now defined in the software instead of by the cold boot operation. These functions are compiled at the time they are required by the software, instead of at cold-boot time, to help speed up cold booting.

Corrections

- The functions `cm:global-min` and `cm:global-max` did not return the correct results in the case where no processors were selected in the previous release. This problem has been corrected.
- The functions `cm:float`, `cm:float-new-size`, `cm:f+`, and `cm:f-` now set the overflow flag properly.
- The function `cm:float-sqrt` now sets the `cm:test-flag` for the case that returns `-0`.
- The functions `cm:float-negate` and `cm:float-abs` have been corrected. Previously, they behaved as though all processors were selected in the case where the destination field was the same as the source field.

**LISP Improvements*

Several error messages have been improved, among them those related to `cube-from-grid-address!!`, `grid-from-cube-address!!`, and `pset-grid!!`.

Corrections

- The function `pretty-print-pvar` no longer affects the context flag.
- When new pvars are allocated, their plists (property lists) are now initialized to `nil`. Previously, the plist of a new pvar was incorrectly initialized to the plist of the last pvar created.
- The function `cond!!` now properly handles `t!!` as an else clause and returns a pvar.
- The function `array-to-pvar-grid` now accepts both general and numeric pvars. (The accompanying family of functions, `pvar-to-array`, `array-to-pvar`, and `pvar-to-array-grid` also accept general and numeric pvars as before.)
- The functions `array-to-pvar` and `array-to-pvar-grid` both accept float dest-pvars. (The accompanying functions, `pvar-to-array` and `pvar-to-array-grid`, also accept float source-pvars, as before.)
- The function `isqrt!!` has been corrected. Previously, the function did not behave correctly for unsigned and signed numbers; it corrupted memory and had the potential to corrupt the stack.

The *LISP Simulator

The functions `scan!!` and `scan-grid!!` previously required, when segment-pvars were used, that the value of the segment-pvar in the first active processor be `T`. This is no longer required, as a value of `T` is now assumed for the first active processor. Also, these functions have been sped up considerably when used with common functions like `+!!`, `and!!`, or `copy!!`.

Several additions in functionality have been made to the *LISP Simulator. The simulator now has all the functionality ascribed to the hardware version of *LISP in **LISP Release Notes: Connection Machine Software Release 11*:

- The functions `*pset`, `*pset-grid` and `*pset-grid-relative` now take the optional *collision-mode* and *notify-pvar* arguments. The value `:no-collisions` may now be specified as a combiner argument.
- The functions `pref!!`, `pref-grid!!` and `pref-grid-relative!!` now take the optional *collision-mode* argument.
- The functions `scan!!` and `scan-grid!!` now support the `:include-self` keyword argument.
- The functions `array-to-pvar-grid` and `pvar-to-array-grid` are now supported correctly in the simulator.

Simulator Corrections

- The function `rot!!` now correctly returns a pvar. Previously, it returned `nil`.
- The function `scan-grid!!` previously did not behave correctly when used with a restricted currently selected set of processors. The function has been completely rewritten and now properly handles a restricted currently selected set.
- The function `rot!!` had a `print` statement removed from its body, and now correctly returns a pvar. Previously, it returned `nil`.
- The functions `load-byte!!` and `deposit-byte!!` now explicitly trap on certain error conditions, such as a negative field length. Previously they waited for COMMON LISP to handle the error, which the Lucid implementation does not do.

***Lisp Release Notes**

Version 4.0

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, June 1987

The information in this document is subject to change without notice and should not be construed as a commitment by Thinking Machines Corporation. Thinking Machines Corporation reserves the right to make changes to any products described herein to improve functioning or design. Although the information in this document has been reviewed and is believed to be reliable, Thinking Machines Corporation does not assume responsibility or liability for any errors that may appear in this document. Thinking Machines Corporation does not assume any liability arising from the application or use of any information or product described herein.

Connection Machine is a registered trademark of Thinking Machines Corporation.

Paris and *Lisp are trademarks of Thinking Machines Corporation.

Symbolics and Symbolics 3600 are trademarks of Symbolics, Inc.

VAX and ULTRIX are trademarks of Digital Equipment Corporation.

Copyright © 1987 by Thinking Machines Corporation. All rights reserved.

Thinking Machines Corporation
245 First Street
Cambridge, MA 02142-1214
(617) 876-1111

Contents

1. Major New Features	1
2. The *Lisp Compiler (Field Test Version)	3
2.1 Enabling the Compiler	3
2.2 The Need for Type Declarations	3
2.3 Compiler Options	5
2.4 Safety Optimization and Overflow	10
2.5 Paris Code Generated	11
2.6 Functions Not Yet Compiled	12
3. Other Enhancements	14
3.1 Functionality	14
Scan Functions	14
Block Data Transfer between Arrays and Pvars	17
User-Defined Initialization Lists	19
New Functions	19
New Keyword Values for pretty-print-pvar	21
Multiple Processor Reads	21
Processor Write Notification	23
3.2 Programming	23
Declaring Pvar Types	23
Syntax of Declarations	23
Example Declarations	25
Interfacing Paris Code to *Lisp	25
*Lisp Memory Management—Stack and Heap Storage	26
3.3 Enhancements to the *Lisp Simulator	28
3.4 An Unsupported Feature: Defining Structures	28
4. Software Error Corrections	31
4.1 *Lisp Interpreter Corrections	31
4.2 *Lisp Simulator Corrections	31
5. Restrictions and Warnings	33
5.1 *Lisp Interpreter Restrictions	33
5.2 *Lisp Simulator Restriction	33

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, the record of a backtrace or other error-tracing operation, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142-1214
-------------------	--

Internet Electronic Mail:	customer-support@think.com
--------------------------------------	----------------------------

Usenet Electronic Mail:	ihnp4!think!customer-support
------------------------------------	------------------------------

Telephone:	(617) 876-1111
-------------------	----------------

For Symbolics users only:

The Symbolics Lisp machine, when connected to the Internet network, provides a special mail facility for automatic reporting of Connection Machine system errors. When such an error occurs, simply press CTRL-M to create a report. In the mail window that appears, the To: field should be addressed as follows:

To: bug-connection-machine@think.com

Please supplement the automatic report with any further pertinent information.

These Release Notes apply to the *Lisp language software component of Connection Machine System Software Version 4.0. These Notes amend the current *Lisp documentation, *The Essential *Lisp Manual*, Release 1.7, published July 1986. All update information for *Lisp since Release 1.7 appears here, including *Lisp information previously published in the Release Notes to Connection Machine Software Release 11 (November 1986).

Margin bars indicate the *Lisp update information that is new since Connection Machine Software Release 11.

The notation conventions used for describing *Lisp functions are similar to those used in *Common Lisp: The Language* by Guy L. Steele, Jr. The default values for keyword arguments are shown enclosed with the keyword in italicized parentheses. These parentheses are not to be typed when supplying a keyword value in a call to the function.

1. Major New Features

With CM System Software Version 4.0, the *Lisp programming language now includes the following major enhancements:

- Ability to run on Digital Equipment Corporation's VAX ULTRIX (within a Lucid Common Lisp environment), as well as on a Symbolics 3600-series Lisp machine.
- A compiler. See Section 2 for a description of the *Lisp Compiler.
- Scan functions, which, using a user-specified binary associative function, combine values in all the processors preceding each processor. See Section 3.1.
- Block data transfer between arrays and pvars, which permits more efficient transfer of data between the Connection Machine and its front-end computer. See Section 3.1.
- Support for automatic execution of user-defined forms before or after the Connection Machine is booted. See Section 3.1.
- Other new functions, and new keywords for `pref!!` and related functions, including keywords to efficiently handle the reading of a value from a processor by more than one other processor. See Section 3.1.
- Pvar type declarations, which enhance program execution considerably and allow more precise management of user memory, and the ability to embed Paris instructions within *Lisp programs. See Section 3.2.
- Updates to the *Lisp simulator, which remove all discrepancies between its functionality and the functionality of the *Lisp interpreter. See Section 3.3.
- Correction of several errors in the implementation of *Lisp and the *Lisp simulator. See Section 4.

There are a few restrictions on the use of *Lisp, Version 4.0. These are:

- The *Lisp of CM System Software Version 4.0 is not backward compatible with CM Software Release 11.
- There are also several known implementation errors in *Lisp Version 4.0. See Section 5.

ERRATUM

Throughout these **Lisp Release Notes*, all references to the function `proclaim` should be read as `*proclaim`.

2. The *Lisp Compiler (Field Test Version)

As of this release, the *Lisp compiler joins the interpreter and simulator to comprise the *Lisp language software component of Connection Machine System Software Version 4.0. The *Lisp compiler in this release is a field test version. The *Lisp interpreter and simulator are not field test versions, and do run on both the Symbolics and the VAX ULTRIX front ends.

The *Lisp compiler compiles *Lisp code into Paris instructions. Compiled code has less stack overhead and data movement than interpreted code, and executes significantly faster. This section describes how to use the compiler, explains the various compiler options, gives examples of Paris code generated by the compiler, and indicates which *Lisp code it currently compiles and which code it must pass to the interpreter for translation into Paris.

2.1 Enabling the Compiler

The compiler is enabled by setting compiler options (see “Setting Compiler Options” below); it is off by default. The compiler, when enabled, executes automatically as part of the Common Lisp compiler. Functions such as `compile`, and `compile-file`, and editor commands such as `Meta-X Compile Buffer` may be used.

2.2 The Need for Type Declarations

The compiler works on pvars where the compiler knows the types of the pvars because of type declarations. Currently, boolean, field, signed, and float pvars are supported. General pvars are not supported. When a *Lisp expression contains general or undeclared pvars, the compiler ignores the expression and lets the interpreter deal with the expression.

Currently, only `*set`, `*max`, `*min`, `*sum`, `*or`, `*and`, `*logior`, and `*logand` expressions are compiled. Also, the predicate for `*when`, the initial values for `*let`, and `*let*` variables are compiled. Pvar expressions appearing elsewhere will not be compiled, but rather will be executed by the interpreter. For example, in the following, the `+!!` pvar expression is not compiled, but both `*!!` pvar expressions are:

```
(proclaim `(type (pvar (unsigned-byte 8)) field))
(*let ((foo (*!! field field)))
  (declare (type (pvar (unsigned-byte 32)) foo))
  (*set-foo (*!! foo foo))
  (+!! foo foo))
```

The use of declarations is critical to the performance of the compiler. The Common Lisp forms `proclaim`, `declare`, and `the` may be used to communicate

type information to the compiler. The compiler is free to use all information from declarations. If a declaration is changed, all code that depends on that declaration must be recompiled. Giving an incorrect declaration is considered an error. The results of an incorrect declaration are unpredictable. The Common Lisp form `deftype` may be used to define a new type. The following `def-type` defines and uses a new type that is equivalent to unsigned pvars of length eight.

```
(deftype char-pvar () `(pvar (unsigned-byte 8)))
(proclaim `(type char-pvar foo))
```

Certain pvar type declarations allow an arbitrary expression to be used for the length. The expression should be simple, should depend only on global variables, should not have side effects, and must compute the length consistently. The compiler arbitrarily generates references to these expressions. The expression may compute different values based on boot time variables. For example, the following defines an unsigned pvar that depends on the cube address length:

```
(proclaim `(type (pvar (unsigned-byte cm:*cube-address-length*))
                  addresses))
```

Currently there are restrictions on where `declare` statements can be placed to be seen by the *Lisp compiler. The compiler can see declarations placed in `*defun`, `*let`, and `*let*`. The compiler cannot see declarations in other locations. For example:

```
(*defun bax (field)
  ;; This declaration is seen by the compiler
  (declare (type (field-pvar 8) field))
  (let ((a 5.0))
    ;; This declaration is not seen by the compiler, so that
    ;; the (!! a), in the following *let will not be compiled.
    (declare (type single-float a))
    (*let ((pvar (+!! field (!! a))))
      ;; This declaration is seen by the compiler.
      (declare (type (pvar single-float) pvar))
      pvar)))
```

Note that the compiler must know the type of the argument to the `!!` function. For this purpose, Common Lisp types such as `single-float`, `double-float`, `integer`, `unsigned-byte`, `signed-byte`, `bit`, and `boolean` may be used. For example:

```
(proclaim `(type float-pvar var))
(*set var (+!! var (!! (the single-float expression))))
```


The following types of proclamations are useful:

```
(proclaim '(type ...))
(proclaim '(optimize ...))
(proclaim '(*optimize ...))
(proclaim '(ftype ...))
(proclaim '(function ...))
```

The optimize proclamation may be used to set the optimization levels safety, speed, space, and compilation speed. The *optimize proclamation has the same effect on the *Lisp compiler as the optimize proclamation, but it does not effect the underlying Lisp implementation. The function and ftype proclamations may be used to give the type of commonly used functions. For example:

```
(proclaim '(ftype (function (t) boolean) willow))
(*set b1 (!! (willow 0.0)))
```

is entirely equivalent to:

```
(*set b1 (!! (the boolean (willow 0.0))))
```

2.3 Compiler Options

A number of options can be set by the programmer to control the behavior of the *Lisp compiler. These options may be changed either by invoking and using the options menu, or by changing the options' associated *Lisp variables directly in code.

To use the options menu do one of the following:

- From a Lisp listener, type:
:Set Compiler Options
- From the editor, press Meta-X and type:
Set Compiler Options
- Execute the following function:
(compiler-options)

To selectively change the value of an option for a region of code, use the Common Lisp special form `compiler-let`. The following example enables the compiler with a safety level of 0 for the region of code enclosed by the `compiler-let` form. (The variables associated with the options are listed with the options' descriptions below.)

```
(compiler-let ((*compilep* t) (*safety* 0))
  (code to compile at low safety))
```

Invoking the options menu produces a window of the following form:

Starlisp Compiler Options

Compile Expressions (Yes, or No)
 Warning Level (High, Normal, None)
 Inconsistency Reporting Action (Abort, Error, Cerror, Warn, None)
 Safety (0, 1, 2, 3)
 Space (0, 1, 2, 3)
 Speed (0, 1, 2, 3)
 Compilation Speed (0, 1, 2, 3)
 Peephole Optimize Paris (Yes, or No)
 Check for Stack Overflow (Yes, Safe, No)
 Add Declares (Everywhere, Yes, No)
 Optimize Bindings (Yes, or No)
 Pull Out Common Address Expressions (Yes, or No)
 Print Length for Messages (an integer, or Nil)
 Print Level for Messages (an integer, or Nil)
 Constant Fold Pvar Expressions (Yes, or No)
 Use Percent Instructions (Yes, or No)
 Percent Instructions Enable Delayed Assembly (Yes, or No)
 Use Always Instructions (Yes, or No)
 Use Maybe Instructions (Yes, Maybe, No)
 Use Undocumented Paris (Yes, or No)
 Machine Type (Compatible, CM1, CM2)
 Use FP Instructions (Yes, or No)

Compile Expressions

This option enables or disables the *Lisp compiler. A value of Yes (t) causes the *Lisp compiler to compile *Lisp expressions. A value of No (nil) disables the *Lisp compiler.

Default: No

Variable: *compilep*

Warning Level

This option controls the type of warnings produced by the *Lisp compiler. A value of High (:high) causes the compiler to generate warnings when an expression was not compiled. A value of Normal (:normal) causes the compiler to generate warnings for invalid arguments and type mismatches. A value of None (:none) prevents the compiler from generating any warnings.

Default: High

Variable: *warning-level*

Inconsistency Reporting Action

This option controls the behavior of the compiler when an inconsistency is discovered. A value of `Abort (:abort)` causes the compiler to report the inconsistency and immediately abort the compilation. A value of `Error (:error)`, `Cerror (:cerror)`, or `Warn (:warn)` causes the compiler to report the inconsistency using the Common Lisp function `error`, `cerror`, or `warn`. `Error` signals a fatal error, from which it is impossible to continue, and enters the debugger. `Cerror` signals a continuable error and enters the debugger, allowing the program to be continued after the error is resolved. `Warn` prints an error message, but normally does not enter the debugger. A value of `None (:none)` prevents the compiler from taking any action.

Default: `Warn`

Variable: `*inconsistency-action*`

Safety

This option controls what kind of code the compiler generates to detect overflow error conditions, and how the error condition is reported. At low safety (0), overflow conditions are not signaled. At higher safety (1), overflow conditions are signaled, but the error message is not very informative and might not occur near the expression that caused the overflow. At highest safety (2 or 3), overflow conditions are signaled, with an error that attempts to be as useful as possible. Low safety, in general, produces the fastest and most dangerous code. See also the section "Safety Optimization" below.

The safety option may also be changed within a program by using either of the following:

```
(proclaim '(optimize (safety value)))
(proclaim '(*optimize (safety value)))
```

Default: `1`

Variable: `*safety*`

Space

This option (potentially) causes the compiler to generate different code when there is a space tradeoff to be made. The choices 0 through 4 have the same meaning as they do for the Common Lisp compiler options.

Default: `1`

Variable: `*space*`

Speed

This option (potentially) causes the compiler to generate different code when there is a speed tradeoff to be made. The choices 0 through 4 have the same meaning as they do for the Common Lisp compiler options.

Default: `1`

Variable: `*speed*`

Compilation Speed

This option (potentially) causes the compiler to generate different code faster. The choices 0 through 4 have the same meaning as they do for the Common Lisp compiler options.

Default: 1

Variable: `*compilation-speed*`

Peephole Optimize Paris

This option controls the *Lisp compiler's optimization of the generated Paris code. A value of Yes (t) causes the *Lisp compiler to optimize the Paris code it generates. A value of No (nil) disables the peephole optimizer. Using the default value may generate unpredictable results.

Default: Yes

Variable: `*optimize-peephole*`

Check for Stack Overflow

This option controls whether or not the compiler generates code to check for stack overflow. A value of Yes (t) causes the check to be made always. A value of Safe (:safe) causes the check to be controlled by the safety level. A value of No (nil) prevents the stack overflow check from being made.

Default: Yes

Variable: `*check-stack*`

Add Declares

This option controls whether or not the compiler generates code with type declarations. A value of Everywhere (:everywhere) causes declare and the forms to be added. A value of Yes (t) causes declare forms to be added. A value of No (nil) prevents declare forms from being added.

Default: No

Variable: `*add-declares*`

Optimize Bindings

This option causes the compiler to generate fewer temporary bindings. A value of Yes (t) causes extra bindings to be removed. A value of No (nil) disables this optimization.

Default: Yes

Variable: `*optimize-bindings*`

Pull Out Common Address Expressions

This option causes the compiler to do common sub-expression elimination on address expressions such as calls to `pvar-location`. A value of Yes (t) enables this optimization. A value of No (nil) disables this optimization. This option should generally not be used as it is not fully implemented. However, it can increase performance significantly in certain circumstances.

Default: No

Variable: `*pull-out-subexpressions*`

Print Length for Messages

Print Level for Messages

These options control the amount of an expression the compiler prints when generating a warning. The Common Lisp variables `print-length` and `print-level` are bound to these variables when messages are printed.

Default: 4

3

Variable: `*slc-print-length*` `*slc-print-level*`

Constant Fold Pvar Expressions

This option causes the compiler to constant fold pvar expressions. A value of Yes (t) enables this optimization. A value of No (nil) disables this optimization.

Default: No

Variable: `*constant-fold*`

Use Percent Instructions

Use Always Instructions

Use FP Instructions

These options cause the compiler to generate percent instructions, always instructions, or direct floating-point instructions. A value of Yes (t) enables the instructions. A value of No (nil) disables the use of the instructions. Use of the percent instructions allows words to be pushed into the IFIFO faster, and may dramatically improve performance. (The IFIFO is the input first-in first-out instruction queue from the front end to the CM.) None of these options are fully implemented, and they generated undocumented Paris instructions.

Default: No

Variable: `*use-percent-instructions*`
`*use-always-instructions*`
`*use-weitek-instructions*`

Machine Type

This option causes the compiler to attempt to generate CM-1 or CM-2 machine-specific code, or to produce code that is compatible across Connection Machine types. A value of `Compatible (:compatible)` causes the generated code to be compatible across machine types. A value of `CM1 (:cm1)`, or `CM2 (:cm2)` allows the compiler to generate code specific to the machine type.

Default: `Compatible`

Variable: `*machine-type*`

Use Undocumented Paris

This option determines whether the compiler generates undocumented Paris instructions. A value of `Yes (t)` allows the compiler to generate both undocumented and documented Paris instructions. A value of `No (nil)` prevents the compiler from generating undocumented instructions.

Default: `Yes`

Variable: `*use-undocumented-paris*`

2.4 Safety Optimization and Overflow

One of the most important and interesting options is the optimization level for safety. This option controls what kind of code is generated for detecting overflow, and how the overflow error is handled. The rationale for this behavior is that detecting overflow is an expensive operation. In debugged code, it is reasonable, and probably desirable, to lower the safety level. In the following `*set` expression, the `+!!` expression may produce a nine bit result, which cannot fit in the eight bit `*set` destination. When the result is too large, overflow has occurred.

```
(proclaim '(type (field-pvar 8) u8 u8-2))
(*set u8 (+!! u8 u8-2))
```

The code produced by the compiler is

```
(progn (cm:u+ (pvar-location u8) (pvar-location u8-2) 8)
  overflow detection code
  nil)
```

In this case, the overflow detection code produced is one of the following expressions, depending on the safety level at compilation time. The number 66575 is a tag encoding certain information such as the `*Lisp` function causing the overflow.

Safety 0

Nothing.

Safety 1

```
(cm::error-if-location cm:overflow-flag 66575)
```

This function reports that an overflow has occurred.

Safety 2

```
(slc::error-if-location cm:overflow-flag 66575 u8)
```

This function not only reports that an overflow has occurred, but also has enough information to describe the processors that have an error, and to display the values in those processors.

Safety 3

```
(if (plusp (cm:global-logior cm:overflow-flag 1))
  (slc::pvar-error cm:overflow-flag 66575 u8))
```

This is similar to safety 2.

For all *Lisp functions, if the destination field is smaller than the source field an overflow error is reported. The following functions are representative of *Lisp functions that report overflow in other circumstances:

lognot!!

In a *set, the lognot!! expression can overflow an unsigned destination.

ash!!

```
+!!          *!!
-!!          /!!
```

Floating point overflow is possible. For /!! overflow occurs for division by zero.

```
ceiling!!    mod!!
truncate!!   rem!!
round!!
floor!!
```

Overflow occurs for division by zero.

```
sqrt!!
isqrt!!
```

Overflow occurs for negative numbers.

float!!

Overflow can occur when coercing a very large integer to a float-point value, or when coercing a floating-point value to a smaller precision.

pref-grid-relative!!

Overflow occurs when getting a value off of the grid.

2.5 Paris Code Generated

The compiler is implemented as part of macro-expansion of *Lisp macros such as *set. Because of this, it is relatively easy to look at the output code generated by the *Lisp compiler. In the editor, the commands Macro Expand Expres-

sion (C-sh-M) and Macro Expand Expression All (M-sh-M) may be used to expand a *Lisp form. Macroexpand may be called directly on a *set or other form, as in:

```
(print (let ((slc::*compiling* t))
        (macroexpand form)))
```

For example, in the following

```
(proclaim '(type (signed-pvar 16) s16 s16-2))
(proclaim '(type (signed-pvar 8) s8 s8-2))
(proclaim '(type boolean-pvar b1))
```

The following *set expression may be expanded

```
(*set s16 (+!! (*!! s8 s8-2) s16-2))
```

to produce the following generated Paris code. Notice that no stack space is used to compute the result of the *set expression:

```
(progn
  (cm:multiply (pvar-location s16)
               (pvar-location s8)
               (pvar-location s8-2) 16 8 8)
  (cm:+ (pvar-location s16) (pvar-location s16-2) 16)
  (cm:error-if-location cm:overflow-flag 66575)
  nil)
```

The following *sum expression may be expanded

```
(*sum (if!! b1 s8 s8-2))
```

to produce the following generated Paris code:

```
(let* ((slc::stack-index *lisp-i:*stack-index*)
       (if!!-context1 (+ slc::stack-index 8))
       (if!!-index2 (+ if!!-context1 1)))
  (*lisp-i:check-stack if!!-index2)
  (cm:store-context-always if!!-context1)
  (cm:load-context (pvar-location b1))
  (cm:move slc::stack-index (pvar-location u8) 8)
  (cm:logandc1-always cm:context-flag if!!-context1 1)
  (cm:move slc::stack-index (pvar-location u8-2) 8)
  (cm:load-context-always if!!-context1)
  (*lisp-i:with-stack-variables (if!!-context1)
    (cm:global-add slc::stack-index 8)))
```

2.6 Functions Not Yet Compiled

The current field test version of the compiler does not yet compile all *Lisp functions. User functions that cannot be compiled are handled by the inter-

preter. Functions defined by `*defun` are currently not compiled. In addition, the following list of functions are currently not compiled (the compiler prints a message when a function is not compiled):

<code>deposit-byte!!</code>	<code>sin!!</code>
<code>*pset-grid-relative</code>	<code>cos!!</code>
<code>off-grid-border-p!!</code>	<code>log!!</code>
<code>array-to-pvar</code>	<code>array-to-pvar-grid</code>
<code>pvar-to-array</code>	<code>pvar-to-array-grid</code>

The following functions compile with the restriction that the warning level is not set to High:

```
load-byte!!
pref-grid-relative!!
pref!!
pref-grid!!
*pset
*pset-grid
```

The following, although compiled, produce large and rather inefficient expansions:

```
*when
*let
*let*
```

3. Other Enhancements

This section describes several features of *Lisp that are not described elsewhere, including scan functions for combining values across processors, data transfer between the front end and the Connection Machine, automatic initialization when cold- or warm-booting the Connection Machine, pvar type declarations, new keyword values for pretty-print-pvar, and efficient handling of multiple reads of values from single processors in `pref!!` and related functions.

3.1 Functionality

Scan Functions

Two new functions are available, `scan!!` and `scan-grid!!`.

```
scan!! pvar function &key (:direction :forward)           [Function]
      :segment-pvar segment-pvar
      (:include-self t)
```

For each selected processor, the value returned to that individual processor is the result of reducing the pvar values in all the processors preceding it. Its own pvar value is by default included in the reduction as well. “Reducing” in this context refers to the Common Lisp function, `reduce`, which accepts two arguments, *function* and *sequence*. The `reduce` function applies *function*, which must be a binary associative function, to all the elements of the *sequence*. For example, if `+` were the *function* all the elements in *sequence* would be summed. For a `scan!!` function, the sequence becomes the pvar values contained in the ordered set of selected processors.

The `scan!!` argument *function* must be one of the following associative binary *Lisp functions: `+!!`, `and!!`, `or!!`, `max!!`, and `min!!`. In addition, the `copy!!` function is supported. (Arbitrary binary functions which accept two pvar arguments and return a pvar result will be accepted as *function* arguments in a future release.) In the following illustration, `*` is any of these binary functions:

(self-address!!)	processor-selected?	value of pvar	result of scan
0	no	a	
1	yes	b	b
2	yes	c	b*c
3	no	d	
4	yes	e	(b*c)*e
5	no	f	
6	yes	g	((b*c)*e)*g
7	no	h	

If * were the function +!! , this would be a summation over the set of selected processors, ordered by cube address:

```
(self-address!!)           => 0 1 2 3 4 5 6 7 ...
(scan!! (self-address!!) '+!!) => 0 1 3 6 10 15 21 28 ...
```

While the scan functions can be explained in terms of the Common Lisp reduce function, there is one important difference. Unlike reduce, the order in which the binary operation is performed on its operands is not guaranteed. The result is that when scanning floating-point pvars that have values in different processors of widely differing orders of magnitude, precision may be lost. Scans performed on floating-point pvars having values in different processors of similar orders of magnitude are not affected.

Normally, the value returned for the last selected processor is the result of applying the *function* to all the preceding selected processors and the last one. One may however, break up the processors into *segments*. A segment consists of a sequence of processors in ascending cube-address order. A new segment of processors begins at each processor in which *segment-pvar* is non-nil. Even if all segment pvars are nil, however, there is always at least one segment beginning with the selected processor having the lowest cube address. The first processor in a segment always receives the value of *pvar* instead of the reduction of all the preceding processors. For example:

```
(self-address!!)           => 0 1 2 3 4 5 6 7...
segment-pvar               => nil nil nil t t nil nil t
(scan!! (self-address!!) '+!!
 :segment-pvar segment-pvar) => 0 1 3 3 4 9 15 7...
```

In this example there are four segments. The first is 0, 1, 2; second is 3; third is 4, 5, 6; and fourth is 7... .

Unlike the other functions which can be used as arguments, *copy!!* exists only as a scanning *function*. It is used only in conjunction with *segment-pvar*. It will cause the value of *pvar* in the first processor of a segment to be copied into all the other processors of that segment. For example:

```
(self-address!!)           => 0 1 2 3 4 5 6 7...
segment-pvar               => nil nil nil t t nil nil t
(scan!! (self-address!!) 'copy!!
 :segment-pvar segment-pvar) => 0 0 0 3 4 4 4 7...
```

The direction of the scanning is normally from lowest to highest cube-address. If the *:direction* keyword argument is *:backward*, then the scan is from highest to lowest cube-address. When scanning backwards, segments are sequences of processors in descending cube-address order. In this example, the segments consist of first, ...7, 6, 5; next, 4; and lastly, 3, 2, 1, 0.

```

(self-address!!)          =>  0   1   2   3   4   5   6   7...
  segment-pvar            => nil nil nil t   t   nil nil t
(scan!! (self-address!!) '+!!
  :segment-pvar segment-pvar
  :direction :backward)   =>  6   6   5   3   4   18  13  7...

```

Normally, each processor receives the result of applying *function* to all the processors before it and including itself. The `:include-self` keyword controls whether the value of a processor is included. When `:include-self` is `nil`, there are two effects:

1. The value of each processor is the result of applying *function* to all processors before it, excluding itself.
2. The value of processors in which *segment-pvar* is non-`nil` is the result of applying *function* to all the processors of the *previous* segment.

Following are two examples:

```

(self-address!!)          =>  0   1   2   3   4   5   6   7...
  segment-pvar            => nil nil nil t   t   nil nil t
(scan!! (self-address!!) '+!!
  :segment-pvar segment-pvar
  :include-self t)        =>  0   1   3   3   4   9   15  7...
(scan!! (self-address!!) '+!!
  :segment-pvar segment-pvar
  :include-self nil)      =>  *   0   1   3   3   4   9   15...

```

In this first example, the case where `:include-self` is `t` is identical to the first `scan!!` example. The processor of cube address 3 receives its own address added to no others, being the first processor in a new segment. In the second case, where `:include-self` is `nil`, the value of processor 0 is undefined since there are no processors preceding it. Processor 3 receives a value that is the sum of the previous segment's processor addresses, $0 + 1 + 2$. Likewise, processor 7 receives a value that is the sum of the previous segment's processor addresses, $4 + 5 + 6$.

The second example illustrates the double effect achieved when `:include-self` is `nil`, using the `max!!` function:

```

pvar                      =>  1   10  5   20  3   4   5   6
segment-pvar              => nil  nil nil t   t   nil nil t
(scan!! pvar 'max!!
  :segment-pvar segment-pvar
  :include-self t)        =>  1   10  10  20  3   4   5   6
(scan!! pvar 'max!!
  :segment-pvar segment-pvar
  :include-self nil)      =>  *   1   10  10  20  3   4   5

```


Scanning can be accomplished using grid addressing as well as cube addressing.

```
scan-grid!! pvar function &key (:dimension :x) [Function]
              (:direction :forward)
              :segment-pvar segment-pvar
              (:include-self t)
```

The function `scan-grid!!` is similar to `scan!!` except that processors are scanned in grid order instead of cube order. The keyword argument `:dimension` controls whether the scanning is done across rows or columns. It may be one of the symbols `:x` (rows) or `:y` (columns), or it may be a non-negative integer less than `*number-of-dimensions*`. A `:dimension` value of 0 corresponds to rows and a value of 1 corresponds to columns. Each row or column is a separate segment.

Block Data Transfer between Arrays and Pvars

Transferring data between the front-end computer and the Connection Machine may be done much more efficiently when the either source or destination of the transfer is an array. Instead of repetitively calling `pref`, or `setf` on `pref`, portions of the array can be moved in block mode using the functions described below.

```
pvar-to-array source-pvar &optional dest-array [Function]
              &key (:array-offset 0)
                  (:cube-address-start 0)
                  (:cube-address-end *number-of-processors-
                  limit*)
```

This function moves data from *source-pvar* into *dest-array* in cube-address order. If provided, *dest-array* must be one-dimensional. If a *dest-array* is not provided, an array is created of size *cube-address-end* minus *cube-address-start*. The data from *source-pvar* in processors *cube-address-start* through *1 - cube-address-end* are written into *dest-array* elements starting with element *array-offset*. The result returned by `pvar-to-array` is *dest-array*.

```
array-to-pvar source-array &optional dest-pvar [Function]
              &key (:array-offset 0)
                  (:cube-address-start 0)
                  (:cube-address-end *number-of-processors-
                  limit*)
```

This function moves data from *source-array* to *dest-pvar*. The *source-array* must be one-dimensional. The other arguments behave the same way as in `pvar-`

to-array. If a *dest-pvar* is not provided, *array-to-pvar* creates a destination pvar, in which case the call to the function must be within a function that accepts pvar expressions, such as **set*. If a destination pvar is created, its value in processors to which *array-to-pvar* did not write is undefined. The value returned by this function is *dest-pvar*.

```
pvar-to-array-grid source-pvar &optional dest-array      [Function]
    &key (:array-offset
          (make-list *number-of-dimensions*
                    :initial-element 0))
    (:grid-start
      (make-list *number-of-dimensions*
                :initial-element 0))
    (:grid-end *current-cm-configuration*)
```

This function moves data from *source-pvar* into *dest-array* in grid address order. If provided, *dest-array* must have the same number of dimensions as the current Connection Machine configuration. If *dest-array* is not specified, an array is created with dimensions *grid-end* minus *grid-start*, where the subtraction is done component-wise to produce a list suitable for *make-array*. The data from *source-pvar* in the sub-grid defined by *grid-start* and *grid-end* as the upper and lower corners, respectively, are written into a similar sub-grid of *dest-array* starting with element *array-offset* as the upper corner. The arguments *array-offset*, *grid-start*, and *grid-end* must be lists of length **number-of-dimensions**. The value returned by *pvar-to-array-grid* is *dest-array*.

```
array-to-pvar-grid source-array &optional dest-pvar      [Function]
    &key (:array-offset
          (make-list *number-of-dimensions*
                    :initial-element 0))
    (:grid-start
      (make-list *number-of-dimensions*
                :initial-element 0))
    (:grid-end *current-cm-configuration*))
```

This function moves data from *source-array* to *dest-pvar* in grid address order. The number of dimensions *source-array* has must be equal to **number-of-dimensions**. The other arguments to this function behave the same way as in *pvar-to-array-grid*. If *dest-pvar* is nil, *array-to-pvar-grid* creates a destination pvar, in which case the call to the function must be within a function that accepts pvar expressions, such as **set*. If a destination pvar is created, its value in processors to which *array-to-pvar-grid* did not write is undefined. The value returned is *dest-pvar*.

User-Defined Initialization Lists

Users can define a set of forms to be executed automatically before and after each execution of `*cold-boot` and `*warm-boot`. These user-defined initialization lists are stored in one or more of these variables:

`*before-*cold-boot-initializations*` [Variable]

`*after-*cold-boot-initializations*` [Variable]

`*before-*warm-boot-initializations*` [Variable]

`*after-*warm-boot-initializations*` [Variable]

New forms are added using the function `add-initialization`, and removed using `delete-initialization`.

`add-initialization` *name-of-form form variable* [Function]

The argument *name-of-form* gives a name to the form being added, and is a character string. The argument *form* may be any executable Lisp form. Adding two forms with the same name is permissible only if the forms are the same according to the function `equal`; otherwise an error is signaled. The *variable* should be one of the initialization-list variables above, or it may be a list of such variables, in which case the *form* is added to each initialization list named. So the *form* and *variable* arguments are not evaluated during the call to `add-initialization`, each argument must be preceded by a single quote. For example:

```
(add-initialization (string 'items)
                   '(initialization-items)
                   '*after-*warm-boot-initializations*)
```

`delete-initialization` *name-of-form variable* [Function]

This operation deletes the form named by *name-of-form* from the initialization list (or lists) specified by *variable*. The arguments are specified in the same manner as the first and third arguments for `add-initialization`. For example:

```
(delete-initialization (string 'items)
                       '*after-*warm-boot-initializations*)
```

New Functions

`oddp!!` *integer-pvar* [Function]

This predicate is true if the argument *integer-pvar* is odd (not divisible by 2), and otherwise is false. It is an error if the contents of *integer-pvar* is not an integer in some processor.

`evenp!! integer-pvar` [Function]

This predicate is true if the argument *integer-pvar* is even (divisible by 2), and otherwise is false. It is an error if the contents of *integer-pvar* is not an integer in some processor.

`signum!! number-pvar` [Function]

This function returns a pvar containing -1, 0, or 1 according to whether the number is negative, zero, or positive. For a floating-point number, the result will be a floating-point number of the same format.

`float!! number-pvar &optional other-pvar` [Function]

This function converts any number to a floating-point number. In processors in which *number-pvar* already contains floating-point numbers, those numbers are returned; otherwise, single-float numbers are produced. When the optional argument *other-pvar* is given, which must contain floating-point numbers, *number-pvar* is converted to the same format as *other-pvar*.

`rot!! integer-pvar n-pvar word-size-pvar` [Function]

This function returns *integer-pvar* rotated left *n-pvar* bits, or rotated right $|n-pvar|$ bits if *n-pvar* is negative. The rotation considers *integer-pvar* as a number of length *word-size-pvar* bits. This function is especially fast when *n-pvar* and *word-size-pvar* are both constant pvars.

`plusp!! number` [Function]

This predicate returns `t` if the argument *number* is greater than zero, and `nil` otherwise. The argument must be non-complex.

`minusp!! number` [Function]

This predicate returns `t` if *number* is less than zero, and `nil` otherwise. However, `(minusp!! (!! -0.0))` is always false. The argument must be non-complex.

`sin!! radians` [Function]

`cos!! radians` [Function]

The function `sin!!` returns the sine of the argument, `cos!!` returns the cosine, and `tan!!` returns the tangent. The argument is in radians, and may be complex.

`log!! number &optional base` [Function]

This function returns the logarithm of the argument *number* in the base *base*. If *base* is absent, the natural logarithm is returned.

New Keyword Values for `pretty-print-pvar`

*Lisp now includes several new global variables, which serve as keyword arguments to the function `pretty-print-pvar`.

```
pretty-print-pvar pvar [Function]
    &key (:mode *ppp-default-mode*)
          (:format *ppp-default-format*)
          (:per-line *ppp-default-per-line*)
          (:start *ppp-default-start*)
          (:end *ppp-default-end*)
```

```
*ppp-default-mode* [Variable]
```

This variable provides the default value for the keyword argument `:mode`. Its initial value is the keyword `:cube`. Its other legal value is `:grid`.

```
*ppp-default-format* [Variable]
```

This variable provides the default value for the keyword argument `:format`. Its initial value is the string `"~s "`.

```
*ppp-default-per-line* [Variable]
```

This variable provides the default value for the keyword argument `:per-line`. Its initial value is `nil`.

```
*ppp-default-start* [Variable]
```

This variable provides the default value for the keyword argument `:start`. Its initial value is zero.

```
*ppp-default-end* [Variable]
```

This variable provides the default value for the keyword argument `:end`. Its initial value is `*number-of-processors-limit*`, and it is reset to this value whenever a `*cold-boot` is executed.

Multiple Processor Reads

Several functions now take an additional optional argument, called *collision-mode*, that determines how cases are handled for efficiency where more than one processor is reading from a single processor. The order and type of the additional arguments (that is, whether they are optional or keywords) is subject to change in the next release.

`pref!! pvar-expression cube-address-pvar` [Function]
 &optional collision-mode

`pref-grid!! pvar-expression &rest grid-address-pvars` [Function]
 &optional collision-mode
 &key :border-pvar border-pvar

The `pref!!` functions, with the exception of `pref-grid-relative!!`, take the additional optional argument, *collision-mode*. The values allowed are `:collisions-allowed` (the default), `:no-collisions`, and `:many-collisions`. This argument allows *Lisp to optimize calls to `pref!!` in the cases where each address is unique, as in `:no-collisions`, or when many addresses are identical, as in `:many-collisions`.

`:collisions-allowed`

This is the default mode. Each processor may access any other processor and multiple reads are allowed. The time required to complete this operation is proportional to the maximum number of processors reading from a single processor.

`:no-collisions`

This tells *Lisp that no two processors will ever be caused to read from the same processor. It allows the Connection Machine to execute the read significantly faster than the `:collisions-allowed` case does. However, if two processors do attempt to read from the same processor, the behavior is unpredictable. Use with caution!

`:many-collisions`

This is useful when there are many processors reading from a single processor. *Lisp uses a different algorithm to resolve the collisions. The result is that the `pref!!` almost takes constant time regardless of the routing pattern. That time is approximately the same as a delivery cycle using `:collisions-allowed` where thirty processors are reading from a single processor, although this may vary for different virtual processor ratios.

`*pset combiner value-pvar dest-pvar cube-address-pvar` [Function]
 &optional notify-pvar collision-mode

`*pset-grid combiner value-pvar dest-pvar x-pvar y-pvar` [Function]
 &optional notify-pvar collision-mode

The `*pset` functions, with the exception of `*pset-grid-relative`, also take an optional *collision-mode* argument. In this case it may take on only the two values `:collisions-allowed` (the default) or `:many-collisions`,

and `:no-collisions` is specified as a *combiner* argument. As with `pref!!`, the time required to complete a delivery cycle is proportional to the maximum number of messages arriving at a single processor. When that number is greater than about twenty, it is better to specify `:many-collisions`. The cut-over point depends on the type of combiner. The *combiner* argument `:no-collisions` causes messages to be delivered to processors somewhat faster than the other combiners.

Processor Write Notification

A new optional argument to `*pset` and `*pset-grid` is *notify-pvar*. This argument must be a pvar; its value when `*pset` has finished executing is `t` in all processors into which a value is written, even if the value written happens to be the same as the pvar's current value, and is *not affected in other processors*.

Note that the syntax of `*pset-grid` will have to change in a future release because the assumption that there are always exactly two dimensions for grid addressing will be incorrect.

3.2 Programming

Declaring Pvar Types

*Lisp does not require that the programmer declare the type of a pvar, nor does it require that the type of a given pvar's value in each processor be the same. A pvar can have an integer in one processor and a floating-point number in the next. The type name of such a pvar is *general*, and undeclared pvars are assumed by *Lisp to be of type *general*.

*Lisp implements general pvars by storing the type information in the CM processors. As a result, whenever a function involving an undeclared pvar is executed, the front-end computer must determine what types of values the pvar has in different processors by communicating with the CM. If there are several distinct types, the front end must tell the CM how to process each one individually. The overhead generated when *Lisp must assume that a pvar is of type *general* is avoided when pvar types are explicitly declared. *Lisp programs containing pvar type declarations are significantly faster.

Syntax of Declarations

The pvar types supported by *Lisp are signed and unsigned integers, floating-point numbers of varying precision and of arbitrary user-defined precision, and Booleans. Pvar type declarations presently are processed only by `*proclaim`,

let*, **let, **defun*, and *the*. These declarations have the same syntax as those described in the Common Lisp manual. The type of a pvar is specified in the following manner:

(pvar (*element-type length*))

Certain types also have a syntax (*element-type length*), and where the length is known, as with certain floating-point types and Booleans, there is a short-hand syntax, *element-type-pvar* (no parentheses). The options for the *element-type* argument are detailed below. The *length* argument specifies the number of bits of CM memory used to represent the pvar.

(pvar (*unsigned-byte length*)) *or* (*field-pvar length*)

Either of the above two types declares a pvar to contain a positive integer of *length* bits. The minimum allowed length is 1 bit. For example, if *length* is 8, the pvar may contain integers between 0 and 255.

(pvar (*signed-byte length*)) *or* (*signed-pvar length*)

Either of the above two types declares a pvar to contain a signed integer of *length* bits. The minimum allowed length is 2 bits. For example, if *length* is 8, the pvar may contain integers between -128 and 127.

(pvar short-float)	<i>or</i>	short-float-pvar
(pvar single-float)	<i>or</i>	single-float-pvar
(pvar double-float)	<i>or</i>	double-float-pvar
(pvar long-float)	<i>or</i>	long-float-pvar

These types describe floating point numbers of different mantissa and exponent sizes. The short-float, single-float, double-float and long-float types are standard IEEE formats.

(pvar (*defined-float mantissa-length exponent-length*))
(float-pvar *mantissa-length exponent-length*)

It is possible to create a floating point number with an arbitrary number of mantissa and exponent bits through the use of *defined-float* or *float-pvar*. If *mantissa-length* and *exponent-length* are not supplied to *float-pvar*, they will default to the single-float sizes. See *Connection Machine Parallel Instruction Set: The Lisp Interface* for a discussion on minimum and maximum floating-point number sizes and performance considerations.

(pvar boolean) *or* boolean-pvar

Either of the above two types defines a 1-bit pvar that contains either the value *t* or *nil*.

Example Declarations

The following examples illustrate the use of the Common Lisp type and declare statements with the above pvar type declaration statements:

```
(*proclaim '(type (pvar boolean) finished-p))
(*defvar finished-p nil!!)

(*let* ((temp1 (load-byte!! foo (!! 0) (!! 9)))
        (temp2 (sqrt!! temp1)))
  (declare (type (pvar (unsigned-byte 9)) temp1)
            (type (pvar single-float) temp2))
  ;; temp1 may contain values between 0 and 511.
  ;; temp2 may contain single-floats
  ...))

(*defun my-function (dest array-of-single-float-pvars index)
  (declare (type (pvar single-float) dest))
  ;; this declares the result of the aref to a single-float.
  (*set dest (the (pvar single-float) (aref array-of-pvars index))))

(setq xx (allocate!! (!! 1.23) 'xx '(pvar double-float)))
```

*Lisp allows certain elements of declarations to be computed at run time as opposed to compile time. All the *length* arguments to the type declarations may be either constants or run-time expressions such as global configuration variables. Following is a typical example using a global configuration variable:

```
(*let ((temp (self-address!!)))
  (declare (type (pvar (unsigned-byte cm:*cube-address-length*)) temp))
  ...
  body of let)
```

Interfacing Paris Code to *Lisp

It is sometimes necessary to explicitly call Paris instructions from within *Lisp programs. Paris code may be placed anywhere in a *Lisp program as long as the Paris code is placed within a call to the macro `with-paris-from-*lisp`. Ordinarily, the Paris stack is uninitialized and any Paris code that uses the stack will signal an out-of-stack-space error. This macro allocates to the Paris stack the space available in the *Lisp stack, so that Paris may be called. It is not possible to execute *Lisp code within a call to `with-paris-from-*lisp`.

Paris instructions require pvars' memory addresses and lengths as their arguments. While a "pvar" is commonly—and appropriately—regarded as a "parallel

variable", a program variable that has a value in each CM processor, it actually exists in the front end as a Lisp object that points to and describes a field in each CM processor's memory. These fields in the CM contain the values that comprise the parallel variable. To provide pvars' addresses and lengths so they can be used as arguments to Paris instructions, several functions that return this information, which is stored in the Lisp object on the front end, are supplied.

```
(pvar-location pvar)
(pvar-length pvar)
```

These functions return the location (address) and length of a pvar.

```
(pvar-type pvar)
```

This function returns the type of a pvar. (Type information is stored in CM memory.) The type of a pvar may be one of :general, :field, :signed, :float or :boolean. These correspond to pvar types t (for general), unsigned-byte, signed-byte, defined-float, and boolean, respectively.

```
(pvar-mantissa-length pvar)
(pvar-exponent-length pvar)
```

If *pvar* is of type defined-float, then these functions will return the mantissa and exponent lengths.

Following is an example of *Lisp code with embedded Paris instructions:

```
(*let ((cube-address (self-address!!))
      dest
      (source (!! 100)))
  (declare (type (pvar (unsigned-byte cm:*cube-address-length*))
                cube-address)
           (type (pvar (signed-byte 10) dest source)))
  (with-paris-from-*lisp
    (cm:send (pvar-location dest) (pvar-location cube-address)
             (pvar-location source) (pvar-length source))))
```

***Lisp Memory Management—Stack and Heap Storage**

The memory of each Connection Machine processor is broken up into a *stack*, a *heap*, and a *gap*, along with a small number of reserved bits. The space occupied by these blocks is identical in *all* Connection Machine processors and is described by several variables in the front-end computer.

The stack begins near the low end of Connection Machine memory. It grows as necessary into the gap, but may not extend past the **stack-limit**. The **stack-limit** points to the last bit in the gap that may be used by the

stack. The `*stack-index*` always points to the next available bit in the gap. The `*stack-limit*` must always be greater than or equal to the `*stack-index*`.

The stack is used for fast allocation of pvars returned as results of Lisp and *Lisp functions (such as `+!!`), and for storage of some pvars created by `*let`. As with a standard stack, memory is allocated and deallocated on a first in, last out basis.

The heap is used to store all other pvars that may not be allocated and deallocated in the strict order required by a stack. This includes all pvars created by `*defvar`, `allocate!!`, and some pvars created by `*let`.

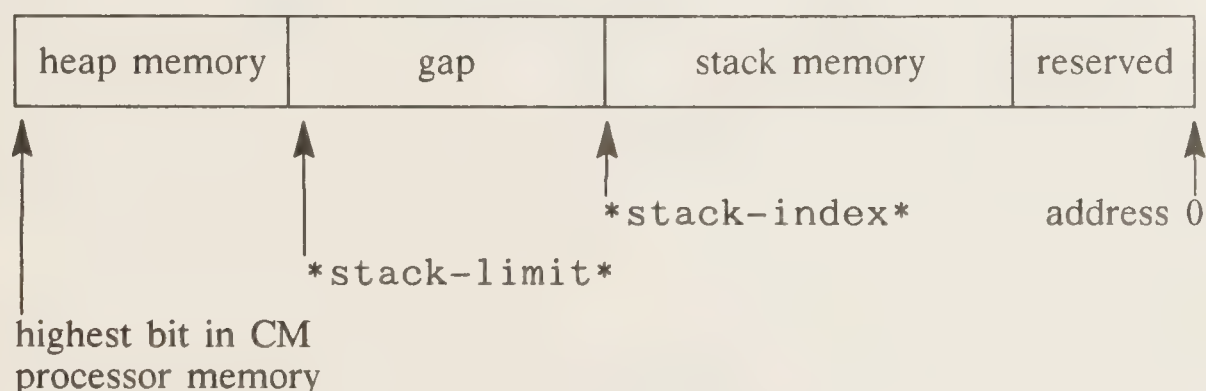


Figure 1. *Lisp Memory Management

Whether a `*let` pvar is allocated on the heap or the stack depends on its type. All general pvars are allocated memory from the heap, while all other types are allocated memory from the stack. In the process of normal computation, general pvars tend to change sizes. This type of memory utilization is more appropriately handled by a heap than a stack.

This is not to say that all general pvars exist in the heap memory. General pvars are returned as the result of a *Lisp function when different processors must contain different types of results. In this case the returned general pvar is allocated on the stack, but it is illegal for it to change size—it must only be used as part of some expression that will eventually be copied into some destination pvar. For example, the following expression is guaranteed to return a general pvar on the stack:

```
(if!! (oddp!! (self-address!!)) (!! 1) t!!)
```


3.3 Enhancements to the *Lisp Simulator

- The *Lisp Simulator now supports all the documented functionality of the *Lisp Interpreter. Specifically, since the previous release, it now supports:
 - The `:include-self` keyword argument in the scan functions. See Section 3.1 for a description of this argument.
 - The functions `pvar-to-array-grid` and `array-to-pvar-grid`. See Section 3.2 for a description of these functions.
 - The optional collision-mode argument for the functions `pref!!`, `pref-grid!!`, `pref-grid-relative!!`, `*pset`, `*pset-grid`, and `*pset-grid-relative`. See Section 3.5 for a description of this argument.
 - The `notify-pvar` argument for `*pset`, `*pset-grid`, and `*pset-grid-relative`. See Section 3.5 for a description of this argument.
- The scan functions have been sped up considerably when used with common functions like `+!!`, `and!!`, or `copy!!`.

3.4 An Unsupported Feature: Defining Structures

At present, the ability to create data types with named record structures and named components is an experimental feature, but it is explained here for the adventurous user. It is very likely this will change.

```
*defstruct (structure-name                                     [Macro]
            &key :conc-name (:immediate-data-type t))
            &rest (element-name initial-value :type pvar-type)
```

This function defines a structure. The keyword argument `:conc-name` behaves the same as Common Lisp's `defstruct` argument of the same name. The argument `:immediate-data-type` currently does nothing, but it must be included with a value of `t` when using `*defstruct`. It may mean something in the future.

Structure objects are created and returned in all selected processors by the following function, which is created automatically when a structure is defined:

```
make-structure-name!!                                           [Function]
```

This function provides keyword arguments for initializing the elements of the structure differently than they are defined in `*defstruct`.

Structures are assigned to pvars using `*set`:

```
*set pvar (make-structure-name!!)
```


Given a *pvar* that contains structures, one may access the individual elements through a set of functions, one for each element, which are created automatically:

element-name!! [Function]

References to the values of structure elements in specific processors are made by another set of functions which are created automatically:

pref-structure-name-element-name pvar address [Function]

The function *setf* is used with *pref* above to modify a structure element.

structurep!! *pvar* [Function]

This predicate checks to see if *pvar* contains any structures; *structurep*!! returns a *pvar* containing *t* in each processor in which *pvar* was a structure, and *nil* otherwise.

The example on the following page defines two structures, an astronaut with age and sex data, and a ship with mass, x-location, and y-location.

- (1) The pvar fleet in odd cube-addressed processors is made to contain the astronaut structure.
- (2) In even cube-addressed processors fleet is made to contain the ship structure.
- (3) The value of the astronaut age in the first processor containing an astronaut is given to astronaut-age-in-proc-1.
- (4) The age in this structure is set to 99.
- (5) The ship mass and x-location in the first processor containing a ship are returned.
- (6) The ship mass in all processors containing the ship structure is set to 22.
- (7) The last line illustrates the structure predicate.

```
(*defstruct (astronaut :immediate-data-type t)
  (age 30 :type (field-pvar 7))
  (sex t :type boolean-pvar))
```

```
(*defstruct (ship :immediate-data-type t)
  (mass 0 :type (field-pvar 5))
  (x-loc -4 :type (signed-pvar 10))
  (y-loc -2 :type (signed-pvar 10)))
```

```
(*defvar fleet)
```

- (1) (*when (oddp!! (self-address!!)) (*set fleet (make-astronaut!!)))
- (2) (*when (evenp!! (self-address!!)) (*set fleet (make-ship!!)))
- (3) (setq astronaut-age-in-proc-1 (pref-astronaut-age fleet 1))
- (4) (setf (pref-astronaut-age fleet 1) 99)
- (5) (pref-ship-mass fleet 0) (pref-ship-x-loc fleet 0)
- (6) (*when (evenp!! (self-address!!))
 (setf (ship-mass!! fleet) (!! 22)))
- (7) (if (*or (structurep!! fleet))
 (format t "There are some structures in fleet"))

4. Software Error Corrections

4.1 *Lisp Interpreter Corrections

- The function `pretty-print-pvar` no longer affects the context flag.
- When new pvars are allocated, their property lists (p-lists) are now initialized to `nil`. Previously, the p-list of a new pvar was incorrectly initialized to the p-list of the last pvar created.
- The function `cond!!` now properly handles `t!!` as an else clause and returns a pvar.
- The function `array-to-pvar-grid` now accepts both general and numeric pvars. (The accompanying family of functions, `pvar-to-array`, `array-to-pvar`, and `pvar-to-array-grid`, also accept general and numeric pvars as before.)
- The functions `array-to-pvar` and `array-to-pvar-grid` both accept float dest-pvars. (The accompanying functions, `pvar-to-array` and `pvar-to-array-grid`, also accept float source-pvars, as before.)
- The function `isqrt!!` has been corrected. Previously, the function did not behave correctly for unsigned and signed numbers; it corrupted memory and had the potential to corrupt the stack.
- Previously, attempting to find the log of 0 using the function `log!!` produced an incorrect result, and attempting to find the log of a negative number produced an ambiguous error message. These problems have been corrected; if `log!!` is supplied a non-positive number an error is signaled.
- Previously, the function `pref` occasionally returned a single precision floating-point number when the argument supplied was a double precision floating-point pvar. For example, `(pref (!! 1.0d0) 0)` returned `1.0s0`. This problem has been corrected. If the pvar argument is of greater than single precision, a double precision value is returned.
- Previously, division by zero produced an unreadable proceed option. This problem has been corrected.
- Several error messages have been improved, among them those related to `cube-from-grid-address!!`, `grid-from-cube-address!!`, and `pset-grid!!`.

4.2 *Lisp Simulator Corrections

- When new pvars are allocated, their property lists (p-lists) are now initialized to `nil`. Previously, the p-list of a new pvar was incorrectly initialized to the p-list of the last pvar created.

- The function `cond!!` now properly handles `t!!` as an else clause and returns a pvar.
- The functions `scan!!` and `scan-grid!!` previously required, when segment-pvars were used, that the value of the segment-pvar in the first active processor be `t`. This is no longer required, as a value of `t` is now assumed for the first active processor.
- The function `scan-grid!!` previously did not behave correctly when used with a restricted currently selected set of processors. The function has been completely rewritten and now properly handles a restricted currently selected set.
- The function `rot!!` had a print statement removed from its body, and now correctly returns a pvar. Previously, it returned `nil`.
- The functions `load-byte!!` and `deposit-byte!!` now explicitly trap on certain error conditions, such as a negative field length. Previously they waited for Common Lisp to handle the error, which the Lucid implementation does not do.

5. Restrictions and Warnings

- The `*lisp-cl` and `*lisp-zl` packages no longer exist. Programmers should use the `*lisp` package.
- Programmers should not depend on the mapping between grid and cube addresses, as this may change.

5.1 *Lisp Interpreter Restrictions

- When the function `scan-grid!!` is given a double precision floating point pvar as an argument, the pvar it returns contains single precision values.
- The diagnostic `test-*lisp`, when run in Lucid Lisp on the VAX ULTRIX front end, produces a small number of errors that do not appear when `test-*lisp` is run on a Symbolics Lisp machine. This is because certain errors are not caught; for example the function `mod` does not catch an attempt to divide by zero, as in `(mod 8 0)`. Since the Symbolics Lisp `mod` function does trap this and similar errors, and `test-*lisp` depends on this, these errors do not appear when `test-*lisp` is run in Symbolics Lisp.

5.2 *Lisp Simulator Restriction

- The *Lisp Simulator supports only the pvar type `general`; it does not support any of the other pvar types.

Connection Machine System Software

Release Notes

Version 4.1
CM-2 Field Test Release

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, October 1987

The information in this document is subject to change without notice and should not be construed as a commitment by Thinking Machines Corporation. Thinking Machines Corporation reserves the right to make changes to any products described herein to improve functioning or design. Although the information in this document has been reviewed and is believed to be reliable, Thinking Machines Corporation does not assume responsibility or liability for any errors that may appear in this document. Thinking Machines Corporation does not assume any liability arising from the application or use of any information or product described herein.

Connection Machine is a registered trademark of Thinking Machines Corporation.
Paris, *Lisp, CM-1, and CM-2 are trademarks of Thinking Machines Corporation.
Symbolics, Symbolics 3600, and Genera are trademarks of Symbolics, Inc.
VAX and ULTRIX are trademarks of Digital Equipment Corporation.

Copyright © 1987 by Thinking Machines Corporation. All rights reserved.

Thinking Machines Corporation
245 First Street
Cambridge, MA 02142-1214
(617) 876-1111

Contents

1. About Version 4.1	1
1.1. Major Features	1
1.2. Overview of Release Notes	2
2. The Connection Machine System	3
2.1. Porting Code	3
2.2. Diagnostics	3
3. Paris: The Lisp Interface	4
3.1. Enhancement	4
3.2. Implementation Changes	4
The cm and cmi Packages	4
NEWS Addressing	4
3.3. Known Restrictions	6
4. The Floating-Point Accelerator	7
4.1. Known Restrictions	7
5. *Lisp	9
5.1. *Lisp Interpreter Known Restrictions	9
5.2. *Lisp Simulator Enhancement	10
5.3. *Lisp Simulator Corrections	10
5.4. Documentation Corrections	10
6. The *Lisp Compiler	11
6.1. Implementation Changes	11
6.2. Documentation Corrections	12
Safety Levels	12
A Compiler Option: Use Paris Macros	13

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, the record of a backtrace or other error-tracing operation, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142-1214
-------------------	--

Internet Electronic Mail:	customer-support@think.com
--------------------------------------	----------------------------

Usenet Electronic Mail:	ihnp4!think!customer-support
------------------------------------	------------------------------

Telephone:	(617) 876-1111
-------------------	----------------

For Symbolics users only:

The Symbolics Lisp machine, when connected to the Internet network, provides a special mail facility for automatic reporting of Connection Machine system errors. When such an error occurs, simply press CTRL-M to create a report. In the mail window that appears, the To: field should be addressed as follows:

To: bug-connection-machine@think.com

Please supplement the automatic report with any further pertinent information.

1. About Version 4.1

Connection Machine System Software Version 4.1 supports a field test release of the newest member of the Connection Machine (CM) family, the CM-2. The CM-2 is faster than the CM-1, has more memory per processor, and offers an optional hardware feature, a floating-point accelerator.

These release notes supplement, but do *not* replace, *Connection Machine System Software Release Notes*, Version 4.0, and **Lisp Release Notes*, Version 4.0.

1.1. Major Features

The functionality of CM System Software Version 4.1 is the same as in Version 4.0, with the following exceptions:

- CM System Software Version 4.1 supports the Symbolics Lisp machine running Genera 7.1 as a front end to the field test CM-2. Version 4.1 does not support the VAX ULTRIX system as a front end to the field test CM-2.
- CM System Software Version 4.1 does not support the CM-1. However, programs written for the CM-1 can be recompiled and executed on the CM-2 with Version 4.1.
- The CM-2 has 64K bits of memory per processor, whereas the CM-1 has 4K bits of memory per processor.
- CM System Software Version 4.1 supports the optional floating-point accelerator.
- Improvements have been made to the field test version of the *Lisp compiler.

1.2. Overview of Release Notes

The remaining sections of these release notes detail the changes in functionality in CM System Software from Version 4.0 to Version 4.1.

Section 2. The Connection Machine System

This section describes how to port code to Version 4.1 and how to correctly interpret certain hardware diagnostic error messages.

Section 3. Paris: The Lisp Interface

This section describes an enhancement to certain functions, two implementation changes that may affect some users, and known restrictions to the use of certain functions.

Section 4. The Floating-Point Accelerator

This section describes the effect of the floating-point accelerator and lists its known restrictions.

Section 5. *Lisp

This section lists *Lisp interpreter known restrictions, a *Lisp simulator enhancement, *Lisp simulator corrections, and corrections to the Version 4.0 documentation.

Section 6. The *Lisp Compiler

This section details the implementation changes in the field test *Lisp compiler and corrects errors in the Version 4.0 documentation.

2. The Connection Machine System

2.1. Porting Code

All *Lisp and Lisp/Paris programs must be recompiled under Version 4.1 to run on the CM-2. Programs must be compiled under Version 4.0 to run on the CM-1.

2.2. Diagnostics

Because of an error in the error reporting by the tests `cm:hardware-test-fast` and `cm:hardware-test-complete`, errors on certain chips are reported to be in the wrong locations. However, the correct location can be determined from the erroneous message.

The CM-2 machine that these release notes accompany is configured as a quarter machine cabinet containing 8K processors, instead of the usual 16K processors. (All the standard configurations are also available in Version 4.1.) The 8K processors are arranged as two 4K-processor machines, each with a sequencer; one is located in the upper right portion of the quarter, and the other is located in the lower left portion.

The location of hardware errors is given by the format *letter-digit1/digit2*, as in A-8/0. The *letter* indicates the physical quarter; in this configuration it is always A. *Digit1* is the matrix board number within the quarter, and *digit2* is the chip number on the board. Both values range between 0 and 31 inclusive. A-8/0 refers to chip 0 on board 8 in the A quarter.

In this release of this particular configuration, the logical matrix boards A-8 through A-15, which are normally located in the *upper* left portion of the quarter, are physically located in backplane slots for A-24 through A-31, which are situated in the *bottom* left portion of the quarter. Therefore, if an error is *reported* for any chip in the range of locations A-8/0 through A-15/31, the error is *actually* located on the chip in the range of physical locations A-24/0 through A-31/31.

To summarize, errors in the range A-0/0 through A-7/31 refer to the boards in the top right section of the cabinet; board 0 is rightmost. Errors in the range A-8/0 through A-15/31 refer to the boards in the lower left section of the cabinet; board 8 is leftmost.

3. Paris: The Lisp Interface

The Lisp interface to the Connection Machine Parallel Instruction Set (Lisp/Paris) is documented in *Connection Machine Parallel Instruction Set (Paris): The Lisp Interface*, Release 2.7, July 1986, and in *Connection Machine System Software Release Notes*, Version 4.0. The information in this section updates those documents.

3.1. Enhancement

The behavior of the Lisp/Paris functions `cm:f+`, `cm:f-`, `cm:f*`, and `cm:f/` when overflow or underflow occurs has been changed. Previously, if overflow or underflow occurred the overflow flag was set and the contents of the destination field were unpredictable. Now, when overflow occurs the overflow flag is set and the contents of the destination field are set to infinity. When underflow occurs the overflow flag is not set and the contents of the destination field are set to zero.

3.2. Implementation Changes

The `cm` and `cmi` Packages

Previously, some undocumented Paris symbols were located in the `cm` package. This has been changed: the `cm` package now contains only documented Paris symbols, and the `cmi` (CM Internal) package now contains undocumented Paris symbols. The use of undocumented Paris is generally discouraged, but those currently using such symbols may continue to use them by adding the package prefix `cmi::` to any undocumented Paris symbols appearing in their code.

NEWS Addressing

In Version 4.1, as in Version 4.0, only two-dimensional grid configurations of the processors are supported in Version 4.1. However, owing to the implementation of n -dimensional NEWS addressing to support processor grids of more than two dimensions in the final CM-2 release, the relationship between cube addresses and NEWS addresses has changed in this field test release. Lisp/Paris programmers who performed their own conversions between cube addresses and NEWS addresses on the CM-1, instead of using the Lisp/Paris functions supplied, must read the following.

The term *NEWS addressing* has taken on a new meaning for the CM-2. It no longer refers to addressing the CM processors only as a two-dimensional

grid of nearest neighbors in the North, East, West, and South directions, although in this field test release of the CM-2 this is in fact the only grid configuration supported. It will be possible in later releases to configure the processors on the CM-2 in a grid of any dimension up to thirty-two. Therefore, the term *NEWS addressing* now refers to addressing the processors as an n -dimensional grid using an address consisting of n numbers, where n is any number between one and thirty-two, inclusive.

On the CM-2, the processors and the communication wires between them are arranged in a twelve-dimensional hypercube, with one chip containing sixteen processors at each node. Each node has twelve wires that attach it to twelve other nodes. The CM-1 is also arranged in this way, but in addition has a separate set of wires that implements the two-dimensional NEWS grid; four wires connect each node to its four neighbors. On the CM-2, all dimensions of NEWS addressing are implemented in a new way on the twelve-dimensional hypercube. A one-dimensional NEWS grid is implemented by using only one of the twelve wires between nodes. Each node has exactly two neighbors. A two-dimensional NEWS grid uses two of the twelve wires; each node is connected to four neighbors. Grids of various dimensions are implemented in the same way, using only the appropriate number of wires.

Implementing the CM-2's n -dimensional NEWS addressing on a twelve-dimensional hypercube requires that each bit position of an address correspond to a communication wire. This is done using a Gray code, a numbering scheme that allows processor address increments of one by changing only one bit in the address. Because NEWS addressing uses a Gray code, the mapping between cube addresses and NEWS addresses on the CM-2 is different from the mapping on the CM-1. User-written code that depends on the simple CM-1 mapping between NEWS and cube addresses must be modified to execute on the CM-2.

Code must, for the CM-2, use the Lisp/Paris functions provided to translate between cube and NEWS addresses. These functions handle the Gray code translation transparently. As noted in *Connection Machine Parallel Instruction Set (Paris): The Lisp Interface*, programmers are encouraged to use the functions provided by Lisp/Paris, rather than writing their own, as the Lisp/Paris functions will always behave correctly regardless of whether the mapping between cube and NEWS addresses changes.

3.3. Known Restrictions

- The arguments to the Lisp/Paris function `cm:float-move-decoded-constant` are not handled properly, as documented in *Connection Machine Parallel Instruction Set (Paris): The Lisp Interface*. However, an understanding of how they are handled allows the use of this function.

Instead of using the arguments *signif-constant*, *expt-constant*, and *sign-constant* as Lisp integers representing a constant floating-point value in the same manner as the values produced by the Common Lisp function `integer-decode-float`, the function moves the low-order bits of each of the arguments into their respective subfields in the floating-point *destination* field. The *sign-constant* must have a value of 0 or 1, the *signif-constant* must have a value between 0 and $2^{23} - 1$, and *expt-constant* must have a value between 0 and 255.

- The behavior of the following Lisp/Paris functions with respect to the test flag is incorrect:

<code>cm:store</code>	<code>cm:store-with-max</code>
<code>cm:store-with-add</code>	<code>cm:store-with-min</code>
<code>cm:store-with-logand</code>	<code>cm:store-with-overwrite</code>
<code>cm:store-with-logior</code>	<code>cm:store-with-unsigned-max</code>
<code>cm:store-with-logxor</code>	<code>cm:store-with-unsigned-min</code>

These operations should unconditionally set the test flag to 1 in every virtual processor that receives one or more messages, and should clear the test flag in every processor that receives no messages. However, the test flag is unaffected by these functions.

- The functions `cm:truncate`, `cm:round`, `cm:floor`, and `cm:ceiling` do not set the overflow flag when the result is too large to fit into the destination.
- The function `cm:float-signum` incorrectly produces a result of -2 when its source value is -0.0.
- The function `cm:float-read-from-processor` returns a single precision value of 0 to the front end when 0.0d0 is read.

4. The Floating-Point Accelerator

- The optional floating-point accelerator on the CM-2 causes the following Lisp/Paris functions to operate significantly faster:

<code>cm:f+</code>	<code>cm:f*</code>	<code>cm:float-sqrt</code>
<code>cm:f-</code>	<code>cm:f/</code>	<code>cm:global-float-add</code>

- The higher the virtual processor ratio, the less time is taken per virtual processor to perform floating-point operations. Virtual processor ratios of 2×1 and greater provide better performance.

4.1. Known Restrictions

- The floating-point accelerator currently supports only single-precision arithmetic.
- The following precision problems are caused by a hardware error in the floating-point accelerator and will be fixed in a future CM-2 release:
 - In one-quarter of all floating-point multiply operations, for a random distribution of significands, the Lisp/Paris function `cm:f*` produces results that are incorrect by one bit in the least significant bit of precision. The error is caused by truncation toward zero instead of rounding toward the nearest number. This slight loss of precision is increased with many successive floating-point multiply operations.
 - The Lisp/Paris functions `cm:f/` and `cm:float-sqrt` produce results that are incorrect by one bit in the least significant bit of precision. Both these functions use the floating-point multiply operation to compute their results, and the iterative nature of the implementation algorithms for these functions causes the loss of precision in the least significant bit. For example, when dividing 2.2 by 2.0, `cm:f/` should produce 1.1; instead it produces 1.0999999.

The results given by `cm:float-sqrt` do not always increase as the value of the argument increases. For example, the square root of 1.0000691 is given as 1.0000346, while the square root of 1.0000693 is given as 1.0000345. Out of the $2^{23} = 8,388,608$ single-precision floating-point numbers in the range $1.0 \leq x < 2.0$, the relation $\text{sqrt}(x) \leq \text{sqrt}(x + \text{epsilon})$ fails for 761,609 instances, or 9.08% of the time. In the same range, the maximum absolute error given by the floating-point accelerator between the square of the root and the original value is 1.1920929e-6, as compared to 2.3841858e-7 without the floating-point accelerator.

—The *Lisp functions `truncate!!`, `floor!!`, `ceiling!!`, and `round!!`, when they have two arguments, and the functions `mod!!` and `rem!!`, occasionally produce incorrect results because they use `cm:f/` to compute their results.

The following macro can be defined in Lisp/Paris or *Lisp to selectively turn off the floating-point accelerator for just the functions `cm:f*`, `cm:f/`, and `cm:float-sqrt`:

```
;;; -*- Package: CMI; Mode: LISP -*-

(IN-PACKAGE "CMI")

;;; Pass WITH-FLOATING-POINT
;;;      function: :divide, :multiply, or :sqrt
;;;      mode: :software or :hardware
;;;      form: code to run

(defmacro with-floating-point (function mode form)
  (setq mode (eq mode :hardware))
  (let ((function-symbol (ecase function
                           (:divide `*use-fast-f/*)
                           (:multiply `*use-fast-f**)
                           (:sqrt `*use-fast-float-sqrt*))))
    `(unwind-protect
      (let ((,function-symbol ,mode))
        (update-cs 0 0 t)
        ,form)
      (update-cs 0 0 t))))
```

This macro is used as follows:

`with-floating-point` *function mode form*

The argument *function* is one of `:multiply` for `cm:f*`, `:divide` for `cm:f/`, or `:sqrt` for `cm:float-sqrt`. The argument *mode* is one of `:software`, to turn off the floating-point accelerator, or `:hardware`, to turn on the floating-point accelerator. The argument *form* is any *Lisp or Lisp/Paris form. For example, to turn off the floating-point accelerator for `mod!!`, execute the following:

```
(with-floating-point :divide :software (mod!! a b))
```

To turn on the floating-point accelerator for `sqrt!!`, execute the following:

```
(with-floating-point :sqrt :hardware (sqrt!! a b))
```

5. *Lisp

The *Lisp interpreter and simulator are described in *The Essential *Lisp Manual*, Release 1.7, July 1986, and in **Lisp Release Notes*, Version 4.0. The information in this section updates those documents.

5.1. *Lisp Interpreter Known Restrictions

- For segmented scans, as for non-segmented scans, the floating-point numbers scanned are normalized with respect to the maximum value in the entire pvar, across all segments, instead of with respect to the maximum value within a segment. As a result, the values for scans computed for certain segments—those with values of a much smaller order of magnitude than the maximum—may be lost entirely. Only segments containing values of the same order of magnitude as the maximum value across all segments will have meaningful results.
- The function `scan!!` does not work correctly when the argument *pvar* contains floating-point values and the argument *function* is `+!!`. It produces an error stating it has run out of stack space.
- The function `scan-grid!!` produces incorrect results for unsegmented scans in those processors located in rows whose first processor is not selected. This problem can be avoided by setting the segment pvar to `t` in the first selected processor in the row, whenever the first selected processor in a row is not the first processor in the row.

For example, suppose the value of a pvar in three rows of five processors each is as follows:

```
1  1  1  1  1
1  1  1  1  1
1  1  1  1  1
```

Suppose a summing scan is performed. If all the processors are selected, the correct results are returned:

```
1  2  3  4  5
1  2  3  4  5
1  2  3  4  5
```

Now suppose certain processors are selected, as follows (the value of the context flag is shown; recall that selected processors contain `t`):

```
t  t  t  t  t
nil t  t nil t
t nil t nil t
```

The following incorrect results are returned:

```
1  2  3  4  5
nil 6  7 nil 8
1 nil 2 nil 3
```


The correct results below are returned if the segment pvar is set to t in the second processor from the left in the second row:

```

      1   2   3   4   5
nil  1   2  nil  3
      1  nil  2  nil  3

```

5.2. *Lisp Simulator Enhancement

- For compatibility with the *Lisp compiler, it is possible in the simulator to **proclaim* the compiler option variable **safety**. This has no effect on the behavior of the simulator.

5.3. *Lisp Simulator Corrections

- Previously, the operations **pset*, **pset-grid*, and **pset-grid-relative* produced incorrect results when the arguments *dest-pvar*, *cube-address-pvar*, or *notify-pvar* were the same. This problem has been corrected.
- The operation *proclaim* can no longer be used when proclaiming **defun* functions and **defvar* variables. Instead, the operation **proclaim* must be used.
- Previously, declarations of the form *(type (pvar ...* were not accepted. Any type declaration with *pvar* as its first symbol is now accepted, although in certain circumstances the simulator may issue a warning.

5.4. Documentation Corrections

- In section 3.4, “An Unsupported Feature: Defining Structures,” of the **Lisp Release Notes*, Version 4.0, the following code fragment:

```

(*defstruct (astronaut :immediate-data-type t)
  (age 30 :type (field-pvar 7))
  (sex t :type boolean-pvar))

```

must be replaced with the following code fragment:

```

(*defstruct (astronaut :immediate-data-type t)
  (age (!! 30) :type (field-pvar 7))
  (sex t!! :type boolean-pvar))

```

- Under “Scan Functions” in section 3.1, “Functionality”, of the **Lisp Release Notes*, Version 4.0, it is stated that “The function *scan-grid!!* is similar to *scan!!* except that processors are scanned in grid order instead of cube order.” In addition, for unsegmented scans using *scan-grid!!*, each new row begins a new segment.

6. The *Lisp Compiler

The field test *Lisp compiler is documented in section 2 of **Lisp Release Notes*, Version 4.0. The information in this section updates that documentation.

6.1. Implementation Changes

- The following compiler options have been removed:
 Space
 Speed
 Compilation Speed
 Percent Instruction Enable Delayed Assembly
 Use FP Instructions
- The compiler option Use Percent Instructions has been renamed Use Paris Macros. This option is more thoroughly explained under section 6.2 below than it was in section 2 of *Connection Machine System Software Release Notes*, Version 4.0.
- There is a new value for the compiler option Machine Type, CM2-FPA. This option causes the compiler to generate code specific to the CM-2 with the floating point accelerator.
- The restrictions on compiling the following functions have changed. They now are:

`pref!!`

`pref-grid!!`

The argument *pvar-expression* must be a simple expression, such as a variable.

`pref-grid-relative!!`

The arguments *relative-address-pvars* must be constants such as `(!! x)` where *x* may be a variable or an integer.

`*pset`

`*pset-grid`

The *combiner* argument may not have the value `:default`, and the optional *collision-mode* argument may not have the value `:many-collisions`.

`load-byte!!`

The arguments *position-pvar* and *size-pvar* must be constants such as `(!! x)`, where *x* must be an integer. The integer extracted must have a constant length across the pvar.

- Previously, the *Lisp compiler did not compile the function `off-grid-border-p!!`; it now does.

6.2. Documentation Corrections

Safety Levels

This section replaces the description of the safety levels that appears in section 2.4., "Safety Optimization and Overflow," of the **Lisp Release Notes*, Version 4.0.

Safety 0

Nothing.

If an error occurs it will not be detected. The result of the error-producing operation is undefined.

Safety 1

```
(cmi::error-if-location cm:overflow-flag 66575)
```

This safety level checks for errors, but it does not report an error until the next time a value is read from the CM by *Lisp functions such as `pref`, `*max`, and `*sum`. Because the error is not reported immediately, the normal debugging tools cannot be used to find it.

Safety 2

```
(slot:error-if-location cm:overflow-flag 66575 u8)
```

The behavior of this safety level depends on the value of the variable `*error-if-location*`. If this variable is `t`, Safety 2 behaves like Safety 3; if this variable is `nil`, Safety 2 behaves like Safety 1. This allows greater flexibility for debugging by making a run-time check. For example, code in which you were reasonably confident could be run with this variable set to `nil`, that is, at safety level 1. If, however, it produced an error, the code could be run again with the variable set to `t`, at safety level 3. Reproducing the error using this method does not require recompilation.

Safety 3

```
(if (plusp (cm:global-logior cm:overflow-flag 1))
    (slot:pvar-error cm:overflow-flag 66575 u8))
```

This safety level checks for errors and reports an error immediately, with enough information to describe the processors that have the error and to display the values in those processors.

A Compiler Option: Use Paris Macros

This section replaces the description of a compiler option, Use Percent Instructions, that appears in section 2.3., "Compiler Options," of the **Lisp Release Notes*, Version 4.0. Use Percent Instructions has been renamed Use Paris Macros.

Use Paris Macros

A value of Yes (t) for this option causes the compiler to generate Paris macros. A value of No (nil) disables the use of macros. When macros are not used, Lisp regards each Paris instruction as a call to a function that has been defined with defun. There is a certain amount of overhead associated with making such calls and passing the appropriate arguments. When macros are used, the *Lisp compiler avoids this overhead by replacing each call to a Paris function with the code to execute that Paris function, within the user's program. Paris functions consist of code that pushes the appropriate instructions into the input first-in-first-out (IFIFO) queue from the front end to the CM, to direct the CM's operation. The decreased overhead provided by in-line macro expansion means that instructions to the CM can be pushed more quickly into the IFIFO.

This option may dramatically improve performance where the amount of work performed by the CM, as opposed to the front end, is small. In this case, it is more likely that the CM will complete its work and wait for a period of time before receiving the next instruction from the front end. Since this option causes the front end to speed up its sending of instructions to the CM, it is likely to cause an overall improvement in performance by keeping the CM busier more of the time. This option does not improve the performance of code in which a proportionately high amount of work is performed by the CM over the front end, because in this case it is more likely that the CM stays busy between receiving instructions from the front end.

This option generates undocumented Paris instructions that are not portable between the CM-1 and the CM-2, or between the CM simulators. Furthermore, because the generated code no longer calls Paris functions, debugging may be more difficult. For example, when this option is enabled the compiler currently produces the code below for the following **set* expression:

```
(*set s16 (+!! (*!! s8 s8-2) s16-2))
```


Code produced:

```
(slc::with-paris-macros (t)
  (cmm::cm-multiply
    (pvar-location s16) (pvar-location s8) (pvar-location s8-2)
    16 8 8)
  (cmm::cm++ (pvar-location s16) (pvar-location s16-2) 16)
  (cmi::error-if-location cm:overflow-flag 66575)
  nil)
```

Default: No

Variable: `*use-paris-macros*`

